Name _____

COMP160
General Programming

Laboratory Book
Summer School - 2019

Department of Computer Science

University of Otago

Table of Contents

F.A.Q.

What do I do if I miss a lab session?

Catch up on the lab work outside of your scheduled lab time, either in the COMP160 lab or at home.

How many lab sessions can I miss without failing terms?

You may miss two of your streamed labs before the mid-semester break and two after without failing terms. However, we know that every missed lab session increases your chances of failing the course. If you need to miss a lab session, and haven't completed the current lab work, make sure you catch it up at another time if at all possible.

What if I don't get my lab task finished during the lab session?

Not getting marked will not hurt your terms/attendance record. Get each lab marked when it is finished. Provided it is finished before the "Last chance" date listed on the Timetable page of this lab book you will receive one mark. After this date it will receive half a mark.

If I have a lab marked, do I automatically get terms for that lab?

Not necessarily. Only if it was marked on or before its scheduled time (the Lab No. column of the Timetable). Completing the lab late does not return a missing terms count, though you will get the lab mark (see previous FAQ). **If the scheduled lab was not marked on or before its scheduled lab time the only way of gaining terms for that lab is for you to be logged on to a lab machine during that scheduled lab time.**

How is attendance for terms determined if the current lab is not marked on time?

The server's login records are used to check attendance duration.

How long do I have to stay in the lab to get terms for that lab session?

If you are 2 or more labs behind schedule you will need to be on task for 100 minutes. If you are on track or 1 lab behind then 90 minutes is sufficient from time to time.

Can I come to other lab sessions *as well as* my own?

Yes, as long as you can find a seat that is not needed by somebody streamed to that lab. If the lab gets too full, demonstrators might ask you to leave or wait until another machine becomes available.

Can I come to other lab sessions *instead of* my own?

Yes, but we ask you to come to your streamed session first where at all possible. You might miss out on terms for the current lab if you come after your streamed session. We do not have enough machines for people to turn up randomly.

Can I work at home?

Yes. You are welcome to bring a completed lab to your streamed lab session. BUT if you are working at home and get stuck, STOP and come to the lab to ask for help. We suggest that spending any more than one hour battling over any particular issue is a waste of your time.

What can I do if I am falling behind?

Come to the lab more often. Attend as many lab sessions as you need to catch up.

What do I do if I can't understand the lab task?

Read it through carefully without touching a computer. Try to identify the start-point, the end-point, and the steps required for getting from start to end. Talk to a demonstrator.

How should I prepare for my exam?

Having a good understanding of the concepts used in the lab tasks should put you in a very good position. There will be practise exams on Blackboard. Talk over any concepts you don't understand with a demonstrator.

Can I go over my mid-semester exam with someone?

Yes, and you can get a mark for it. The Lab 17 mark can be achieved by **either** learning to use a Command Line Interface, **or** reviewing your corrections on your mid-semester exam paper with a demonstrator.

If I get over 50% in total for this paper, will I pass the paper?

Not necessarily. The final examination counts 60% towards to your final mark. You need to score at least half marks in the final exam AND have a total mark (including the internal component) of 50% or over to pass the course.
If you do not pass the final exam, even if your total mark (final exam + lab marks + mid-semester exam mark) adds up to over 50% you will receive a Failed Component (FC) grade and therefore fail the course.

Timetable for COMP160

Day	Lec. No.	Lecture	Lab No.	Mark	Lab Name	Last Chance for Full Marks
January 7th	1	Introduction (Chapter 1)				
January 8th	2	Data types and language basics (Chapter 2)	1	1%	Introduction to Java	
January 9th	3	Program structure, methods and basics (Chapter 2)	2	1%	Variables	
January 10th	4	Expressions. Arithmetic (Chapter 2)	3	1%	Methods	Labs 1 and 2
January 11th	5	Graphics, drawing and GUIs (Appendix F)	4	1%	Expressions	Lab 3
January 14th	6	Objects 1 and special methods (Chapter 3)	5	1%	Graphics	Lab 4
January 15th	7	Objects 2. Strings (Chapter 3)	6	1%	Objects	Lab 5
January 16th	8	Structured programming, more maths (Chapter 3)	7	1%	Constructors	Lab 6
January 17th	9	Boolean expressions, blocks, if else (Chapter 4)	8	1%	Math and Random	Lab 7
January 18th	10	Selection (Chapter 4)	9	1%	Selection 1	Lab 8
January 21st	11	Repetition (loops) 1, iterators, iterable (Chapter 4)	10	1%	Selection 2	Lab 9
January 22nd	12	Repetition (loops) 2 (Chapter 4)	11	1%	Strings	Lab 10
January 23rd	13	Objects 3 Classes and methods (Chapter 5)	12	1%	Repetition 1	Lab 11
January 24th	14	Objects 4 References (Chapter 5)	13	1%	Repetition 2	Lab 12
January 25th	**MID-SEMESTER BREAK**					
January 28th	**MID-SEMESTER EXAM**			**15%**	**** material covered in Lecture and Labs 1 to 12**	
January 29th	15	Arrays 1 (Chapter 7)	14	1%	Graphical Objects	Lab 13
January 30th	16	Arrays 2 References to Objects (Chapter 7)	15	1%	Arrays	Lab 14
January 31st	17	Graphics 1 components (Chapter 6)	16	1%	Two-Dimensional Arrays	Lab 15
February 1st	18	Graphics 2 events (Chapter 6)	17	1%	Command Line Interface **OR** Mid-semester exam review *Lab 17 will not be marked after February 1st*	Lab 17
February 4th	19	Graphics 3 examples (Chapter 6)	18	1%	Graphical User Interfaces	Lab 16
February 5th	20	Files input output, sorting (Chapter 10 / readings)	19	1%	Calculator	Lab 18
February 6th	**WAITANGI DAY**					
February 7th	21	Hierarchies, inheritance (Chapter 8)	20	1%	Reading from Files	Lab 19
February 8th	22	Visibility, overriding. (Chapter 8)	21	1%	Shapes 1: Building the Structure	Lab 20
February 11th	23	Hierarchies, abstract classes (Chapter 8)	22	1%	Shapes 2: Animation	Lab 21
February 12th	24	Collections, ArrayList	23	1%	Shapes 3: Abstract	Lab 22
February 13th	25	Simulation. Programming	24	1%	Shapes 4: ArrayLists	Lab 23
February 14th	26	Topics in Computer Science	25	1%	Options (Only for those with 24 lab marks by 3pm Feb 13th)	Lab 24, 25

Introduction

Welcome

Welcome to the laboratory sessions for COMP160. Programming is a very practical skill. The best way to learn is to sit down and write programs. This means that the laboratory sessions are probably the central part of COMP160 and the place that you will really learn the art and craft of computer programming. We hope you find the laboratory sessions useful and enjoyable.

For general course information please visit the course blackboard page (accessed from http://blackboard.otago.ac.nz) regularly.

Please make sure you read these introductory pages. They contain some general information which we will assume that you know from now on.

You are also required to sign page 8 to show that you agree to abide by the rules for use of both the software and the laboratory.

Contact People

Course Coordinator

Professor Anthony Robins

Office : Room 253 , Owheo Building, 133 Union St East

Phone : 479- 8314

Email : anthony@cs.otago.ac.nz

Laboratory

Ms Sandy Garner

Office : Room G11C, Owheo Building, 133 Union St East

Phone : 479-5242

Email : sandy@cs.otago.ac.nz

COMP160 Web Page

http://www.cs.otago.ac.nz/comp160

This site holds preliminary information about the course.

http://blackboard.otago.ac.nz

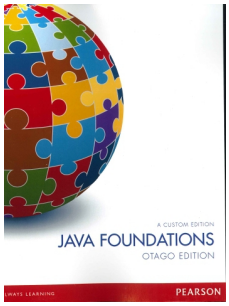
Blackboard is used for notices, study information and clarifications to lab work. Also, files for lab work can be downloaded from the Course Documents link.

About the Laboratory Sessions and Assessment

The Laboratory Workbook (Lab Book)

This book details the practical work for the 5 laboratory sessions per week required for COMP160. (Each laboratory session is two hours long). You need to bring this book with you to each lab session as it makes up part of the work which is marked. We also expect you to have a pen with you.

The Textbook



The text book is:

Java Foundations: Introduction to Program Design and Data Structures

Otago edition

John Lewis, Peter DePasquale, Joseph Chase

Pearson Australia 2015

ISBN 978 1 4886 1128 5

You will need to own a copy, or have ready access to a copy. The labs, including most assessed exercises, are based on the text book (L,D&C). The 2nd, 3rd, 4th and International editions are also fine to use. Page reference numbers given in this book refer to the Otago edition. The International edition is similar in page numbering. If using other editions, the section numbers will be more useful guides than the page numbers.

Labs

Laboratory based work forms 25% of your total assessment for COMP160. There are 25 Labs worth 1% each if you complete them during the specified period.

As you can see from the timetable on page 3, both lectures and labs follow the structure of L,D&C (following the chapters more or less in order).

Each lab consists of three sections:

Notes - The notes may highlight points to watch out for while reading the chapter in L,D&C or provide further relevant background notes.

Preparation - These exercises should be completed **before** the scheduled lab session starts. Preparation will usually involve questions taken from L,D&C or the lecture material, and planning for the program which will be written during the lab session. At the very least, you should read through the lab work and understand the task before arriving in the lab.

Lab Work - This is work that should be completed before or during your lab session. It involves planning and writing a piece of code to perform a specified task.

Assessment

Each lab is marked on an "all or nothing" basis. Preparation, Exercises and Lab Work are assessed. If this work is completed to a satisfactory standard before its "Last Chance for Full Marks" date you get 1 mark for that lab. After that date you will receive 0.5 of a mark. A "satisfactory standard" for Preparation Exercises means that your answers are mostly correct. For Lab Work it means that the programs you write are working, well designed, and include appropriate comments.

When you have completed the work for a lab session your work will be checked, and your lab book will be signed by a demonstrator. Your code should then be submitted electronically to the COMP160 Drop Box on our server. If your laboratory marks are not entered into our records correctly then your signed lab book is proof, along with the files on our servers, that you did complete the lab.

Labs should be marked during your scheduled lab session. If you cannot complete a lab during that time, you should work on it in your own time. Ideally it should be marked **before** the start of your next lab session. If you are working ahead of schedule – great! – but please don't ask the demonstrators for help with material that has yet to be delivered in lectures.

If you haven't completed a lab and had it marked during the lab session it was assigned, your attendance will be recorded for the purposes of Terms by our machine records.

Terms

To gain terms (to be allowed to sit the final exam) you must attend at least 11 of your 13 scheduled lab sessions before the break and 10 of your 12 scheduled lab sessions after the break.

Please do not interpret this as "I only need to complete 21 of the 25 labs". Completing lab work is how your learning happens, and is a separate issue from attending lab sessions. You may miss 4 of your scheduled lab sessions without having to consult or inform. But you should aim to complete the lab work for labs 1 to 24 whether or not you have missed any of your lab sessions.

This 4 session 'freedom' allowance is to allow you to cope with illness, family emergencies and other events that may occur over the semester. Of course, if you use it early without good cause, it will not be available if you need it later.

If you have a legitimate need for absence covering more than 2 labs you can apply to the Teaching Fellow for a terms credit. You will need to provide written proof of your circumstances. The workload of other papers you are studying is not sufficient reason for an extension. Managing course workload is a normal part of every student's university experience.

If you have completed your lab work for the current lab before the end of your lab session, and have had it marked, you do not have to stay for the rest of the session. Submit a copy of your lab code to the COMP160 Drop Box before you leave the lab.

If you have completed your lab work for the current lab before the start of your lab session, and have had it marked, you do not have to attend the session at all.

If you have not completed your lab work by the end of your lab session, your attendance in the lab will be recorded by us using our server log records.

We strongly recommend that you stay for your full 110 minutes for each lab and work ahead on your lab work where possible. This is particularly so for students who have some programming experience e.g. COMP150 and find the first few labs easy. Work steadily ahead until you reach the place where you are challenged.

Work Load

You will find the laboratory time too short to complete the lab work unless you have attended the lectures and done some preparation beforehand.

COMP160 is an eighteen point paper. This means that during Summer School you should expect to spend roughly 25 hours per week working on COMP160. This includes the 15 scheduled hours (5 hours of lectures and 10 hours of laboratories). In other words you have 10 hours per week for your own work on understanding the concepts and preparation for the laboratory sessions. This lab book contains background information and preparation exercises that are designed to be done before each actual laboratory session. It also describes programming exercises to be done during the lab session itself, although you need to think about them **before** you sit down in front of the computer.

Demonstrators may refuse to help you if you haven't completed your preparation exercises, as these are specifically designed to build the knowledge required for your lab work.

The course material builds quickly on itself. Consequently, it is important that you complete and understand the work for one lab session before going on to the next. Some of you may have some programming experience and might feel that the early programming problems are too easy. In this case please try other programming exercises from the text book, or you can work on your own programming projects.

This is a fast moving course – if problems arise it is very important that you ask for help before you fall too far behind. Don't be afraid to ask – we are here to help.

Demonstrators

The demonstrators are there to help you understand the work and help you with any problems. They will give help, but they are not there to simply tell you everything or to write the programs for you! They will try to help you work out your own answers to the exercises and develop your own understanding.

Demonstrating is very difficult work and finding the right balance in answering a question is hard. The process is sometimes frustrating for both them and you, so please be patient and understand the reasons for the amount or kind of help that a demonstrator might give.

Illness

If you are ill / contagious, please stay at home until you are well.

Working together

Many students find it helpful to work in small groups, and this cooperation is to be encouraged – you can learn a lot from each other! Feel free to work on the Preparation questions together with all members of a group taking an **active** part in producing the answer.

Shared Work, otherwise known as Plagiarism

When you present code for marking, it is very important that all of the code is written by you. It does not help you to learn if you simply copy (plagiarise) a written answer or program. In order to pass COMP160 you must pass the final exam, so gaining your own understanding is essential.

- **Do not copy any portion of another student's code.**
- **Do not lend, send or show your code to any other student.** Copying requires two willing persons, the giver and the taker. If another student copies your code, you will lose your marks as well.
- **Do not assume that changing the variable names and comments makes the copying undetectable.** Several previous students can tell you this is not the case.

If you are falling behind and feeling stressed by the pace of the course, copying code is not the best solution.

If you are caught plagiarising the matter will be dealt with severely under the University Regulations. We routinely check your code against that of your fellow students, and previous COMP160 students. In order that we may be able to do this, we get you to deliver a copy of your .java files as you get your labs marked. The demonstrators will show you how to do this when it is required.

Academic Integrity and Academic Misconduct at the University of Otago

Academic integrity means being honest in your studying and assessments. It is the basis for ethical decision-making and behaviour in an academic context. Academic integrity is informed by the values of honesty, trust, responsibility, fairness, respect and courage. Students are expected to be aware of, and act in accordance with, the University's Academic Integrity Policy.

Academic Misconduct, such as plagiarism or cheating, is a breach of Academic Integrity and is taken very seriously by the University. Types of misconduct include plagiarism, copying, unauthorised collaboration, taking unauthorised material into a test or exam, impersonation, and assisting someone else's misconduct. A more extensive list of the types of academic misconduct and associated processes and penalties is available in the University's Student Academic Misconduct Procedures.

It is your responsibility to be aware of and use acceptable academic practices when completing your assessments. To access the information in the Academic Integrity Policy and learn more, please visit the University's Academic Integrity website at www.otago.ac.nz/study/academicintegrity or ask at the Student Learning Centre or Library. If you have any questions, ask your lecturer.

Links to the policies, for your reference:

- Academic Integrity Policy (www.otago.ac.nz/administration/policies/otago116838.html)
- Student Academic Misconduct Procedures (http://www.otago.ac.nz/administration/policies/otago116850.html)

The code I have submitted for this lab is my own work

When you submit each lab for marking, you will be asked to sign a declaration that the code you are submitting is your own work. In full, you are asserting that:

* The lab is my own work.

* I have typed the code myself.

* I have not shared, and will not share, this code with any other student of the current COMP160 class or any class in the future.

* Any help I have received from outside the lab, including that from private tutors, friends and other students, has been about concepts. Nobody else has provided me with lines of code.

* I understand that if my code is similar to another student's, that we may be suspected of plagiarism, and investigated accordingly.

Access to Java Programming Environments

The Owheo labs are 24 hour access, granted using your Otago University ID card and pin number. To install DrJava on another computer:

In order to run, DrJava needs the Java run-time environment to be installed. The latest run-time environment is currently Java 8. At the time of going to print, the url http://www.java.com/en/download/manual.jsp was a good place to begin.

DrJava is available as a free download from `http://www.drjava.org` but there are different requirements for Windows and Mac.

> Windows can use the DrJava Windows App with Java 8.
>
> Mac users need to choose between the Mac OS X App which requires the legacy runtime environment Java 6, and the Jar file which can cope with the current JRE, Java 8. We are using the Jar file in the COMP160 lab this semester.

Sorry, we are unable to provide assistance with your home installation of Java. Please use the help facilities available from the download site.

Rules for use of the COMP160 Laboratory

Software Licences

The software and manuals available in the COMP160 laboratory have been bought under various licences. In general, these licences state that under the Copyright Laws:- No part of this work may be reproduced in any form (in whole or in part) or by any means or used to make a derivative work (such as a translation, transformation or adaptation). Under the law, copying includes translating into another language or format. All rights reserved. Hence, you are NOT permitted to copy the system software or any applications under any circumstances, nor should the systems be used to copy any other licensed or copyright software without the owner's permission. Failure to observe these requirements will be considered a serious breach of University regulations, and will be dealt with under the Discipline Regulations.

Rules

- Your 24 hour access to the Computer Science Department labs is for you and you alone. You must not allow friends to come in to the building with you after-hours.
- You must never allow anyone else to use a departmental computer logged in under your account name. Do not log a friend in and let them work/ play beside you.
- Eating is not allowed while sitting at a computer. You may drink out of sipper top bottles, which are to be kept on the floor.
- Cell phones must be switched to silent and kept off the desk during streamed lab sessions.
- We reserve the right to examine any memory stick or other media brought into the laboratory.
- Students belonging to a particular laboratory session have first priority for the use of the computers and help from the demonstrators in that session. If you want to do extra work and there are no computers available you will have to find another time.
- Large files not related to course work should not be stored in your Home Directory. This includes **mp3**, **movie** and large **graphics** files. Such files may be deleted from your Home Directory without warning.
- Your computer science home directory is to be used responsibly and for coursework only. Its contents can be inspected at any time by departmental staff.

To use the COMP160 Lab you must sign in the box below. By doing so you are accepting notice of these conditions and are agreeing to abide by them.

I have read and agree to abide by the lab rules.

I understand that in order to be allowed to sit the final exam I must attend 21 of my streamed lab sessions.

Name (please write clearly)	**Date**	**Signature**
_____	_____	_____

Laboratory 1 Introduction to Java

A journey of a thousand miles starts with a single step.

– Lao Tzu, Tao Te Ching

Notes

The reading for this lab is Chapter 1 of Lewis, DePasquale and Chase (the text book), particularly Sections 1.1 and 1.2 (pages 2-18). Chapter 1 is very general, and Lab 1 focuses on introducing **DrJava** (the application that we will use to write Java programs). Seldom is a program of any significant length typed in accurately to begin with. This lab explores some of the error messages which DrJava produces to help you correct your code.

On page 15, the textbook describes an Integrated Development Environment (IDE). **DrJava** is a simple IDE which can be used on many different kinds of computer, including Macs, Windows, Linux and other forms of Unix, to develop programs that run on many different kinds of computer. Java programs can also run on many other kinds of device including some cell phones, PlayStations, Palm handhelds, and others. We will use **DrJava** to write and run Java programs on Macs running Mac OS Sierra.

Preparation

In general, preparation questions are taken from the textbook, and you will need to read the textbook in order to answer them. The preparation questions should be completed before you come to the lab session. Note that the textbook contains answers to its Self Review Questions. Working through these may also help you to prepare for your laboratory session. For this first lab, however, there is just one preparation question:

1. How many of your 25 streamed lab sessions must you attend if you want to be permitted to sit the final exam?

Lab Work

In this lab we introduce **DrJava** (the application that we will use to write Java programs) and use it to write a Hello World program. Most programs will contain some errors when they are first typed. We will explore some of the error messages DrJava produces to help you identify the errors.

Part 1

1. Log On to a computer in the laboratory. If you haven't used Mac OS 10 recently, we suggest you take a look at Mac Help (**Finder > Help**) to familiarise yourself with it. You will find it very useful to know how to customise your Dock.

In COMP160, we expect not to have to spell out every detail of a process, but assume that you will be able to find things in file menus, work out which button/key to press to finish a process and make selections in dialog boxes. To call a demonstrator, we ask that you use the **DemoCall** icon on the desktop. This queues your call and ensures people are seen in the order in which they asked for help.

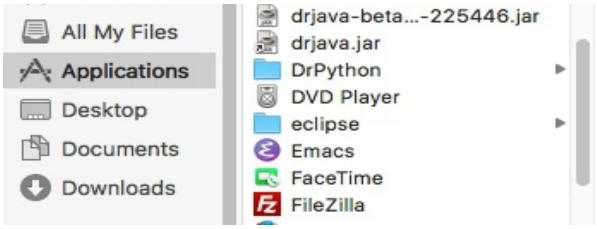
2. Change your password: select **Apple menu> System Preferences> Users & Groups.** Click on the **Change Password** button. Enter your existing password then your new password. Your new password must be **at least 8 characters long** and contain **at least one digit**, **one uppercase character** and **one lowercase character**. Retype your new password in the **Verify** field then select **Change Password**. You will see a message about your login keychain. Select **OK**.

3. Change your password from time to time and keep it secure!

4. You will see the **coursework** folder on the desktop - open it and explore. This folder is stored on our server, not your local machine. In the **COMP160** directory you will find the **coursefiles160** folder which contains any files needed for your lab work. Remember to copy these to your home directory for editing, compiling and running.

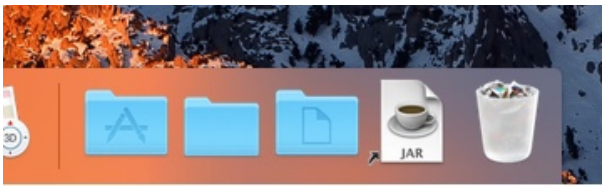
5. In the Computer Science Department laboratories, all your files are stored in your **Home** directory. The word *directory* is interchangeable with the word *folder*. Your **Home** directory, like the **coursework** directory, is stored on our server. You can access it from the Finder window under **Go> Home** or under **Places** at the left hand side

of any Finder window or using the key combination *command-shift-H*. In your **Home** directory there is a **COMP160** directory where your lab work should be saved during the semester.



6. Drag the **drjava.jar** alias (the alias has a small arrow) from the **Applications** directory (**Go > Applications**) to the right hand end of your Dock, next to or near the Trash icon.

   You will only have to do this once. It will remain there for your convenience.



7. Double-click on the Jar file icon to open it. A Java icon will appear on the left side of your Dock and a DrJava window will open on your Desktop.

8. **DrJava** is the application in which you will write and run your Java code. It is an Integrated Development Environment (IDE) providing a graphical user interface (GUI) and tools for developing your computer programs. The IDE allows you to edit, navigate, compile and run your code. You will soon be very familiar with this environment.

9. The physical process required for writing and running a java program involves four steps:

   1) Typing in the code. DrJava will display a large window on the right hand side, underneath the **DrJava** tool bar (Figure 1.1). This is the text editing window into which you will type your code.

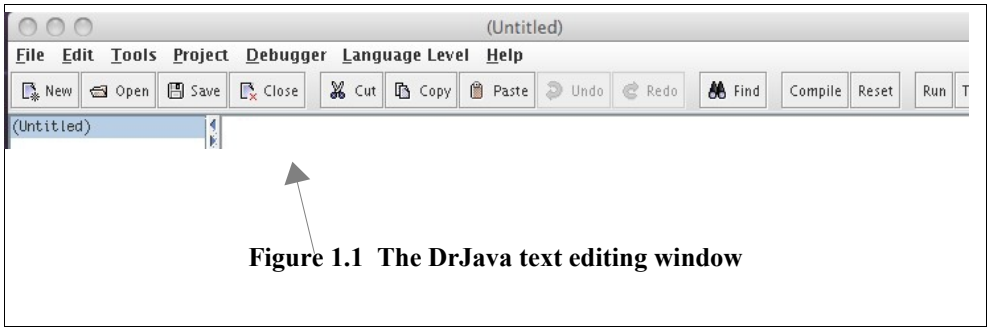


Figure 1.1 The DrJava text editing window

Into the text editing window, type the code for the `HelloApp` class listed below.

```
/** Hello World exercise.
 * Lab 1 COMP160 Part 1
 * (your name) January 2019
 */
public class HelloApp{
   public static void main (String[] args){
      System.out.println("Hello world");
   }
}
```

DrJava will put a space after the star - you will need to remove it

2) Saving your file (class).
Select **File> Save as** and the Save window will appear. Double-click on the COMP160 directory.

Create a new directory for Lab 1 by selecting **New Folder** (Figure 1.2). Name your new folder **Lab1**.

It is possible your Save window will look like this instead. **New Folder** is the middle icon.

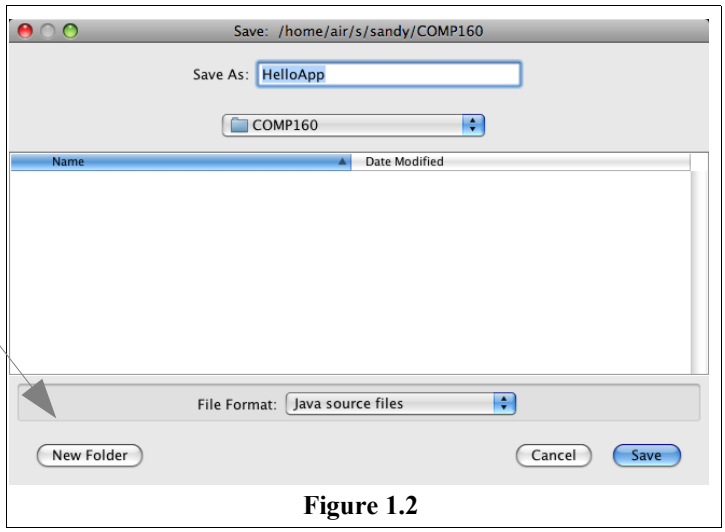


Figure 1.2

A java class needs to be saved in a file of the same name, but with a `.java` extension.

Save your file in the **Lab1** folder with the name `HelloApp.java` (Figure 1.3).

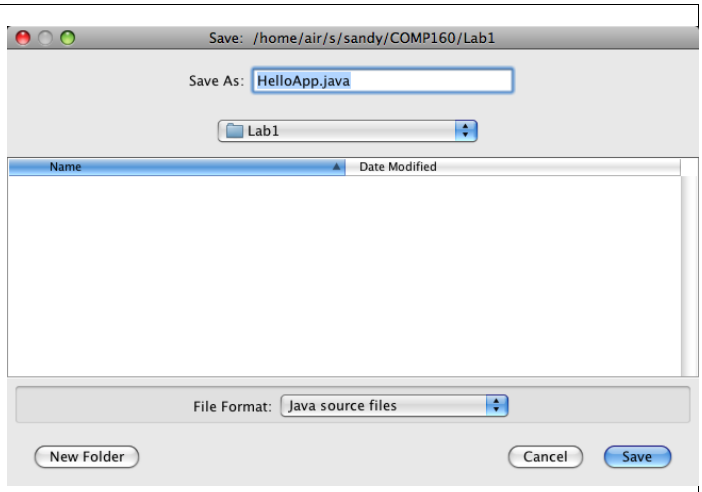


Figure 1.3

3) Compiling your code. Click on the Compile button on the toolbar (or the shortcut `F5` key).

When you see "Compilation completed" in the Compiler Output window, a class file will have been created in your Lab1 directory called `HelloApp.class`. If you see an error message in this window, check your code for typing errors, correct your code, and save again (**File> Save**). Note: Java is case sensitive.

4) Running your program. Run your program using the Run button, or if your version of DrJava doesn't have one, use the **Tools> Run document's main method** (or the (not so short) shortcut `fn` plus `F2` key).

You should see the output of the program in the Console window.

Figure 1.4

Part 2

1. Programming Projects PP 1.1 (L,D&C page 28). In a new `DrJava` document, enter, compile, and run the following application:

```
public class Test{
   public static void main (String[] args){
      System.out.println("An Emergency Broadcast");
   }
}
```

2. Programming Projects PP 1.2 (L,D&C page 29). Introduce the following errors, one at a time, to the `Test` class. Record any error messages that the compiler produces. Fix the previous error each time before you introduce a new one. If no error messages are produced, explain why.

You will find it useful to turn line numbering on in **DrJava** : In **`Preferences> Display Options`** check the box for **`Show All Line Numbers`**

a. change Test to test

b. change Emergency to emergency

c. remove the first quotation mark in the string (in programming, a string is a sequence of characters)

d. remove the second quotation mark in the string

e. change `main` to `man` (this will compile but check the Interactions window when you try to run it)

f. change `println` to `bogus`

g. remove the semicolon at the end of the `println` statement

h. remove the last brace in the program

3. Copy the file called `Ex3App.java` that can be found in the **coursework>COMP160>coursefiles160> Lab01** directory to your **`Lab1`** directory. Open it in `DrJava`. Fix the errors in the code until it produces a correct output.

NOTE that double-clicking, and Open With will **not** open your file. You can drag the `.java` file's icon in to a `DrJava` window or use `File > Open` in `DrJava` to navigate to the file you wish to open.

If you cannot compile your code, make sure you are working on your copy of the file, not the coursefiles copy. We don't allow you to save changes to that one.

Make sure you understand:

* A Java program is made up of one or more class definitions.
* One of the classes must contain a method called `main`. By definition, a class with the `main` method in it is called an application class.
* The `main` method is where statement execution begins.
* Each programming statement in the `main` method is executed (performed) in order until the end of the method is reached.
* The Java language is accompanied by a library of extra classes which are called the standard class libraries.

Lab 1 Completed

The code I present for my Lab 1 mark is my own work according to the definition on page 7 _____

☐ files in Lab1 folder ☐ Part 1 Hello World ☐ Part 2 Error messages
☐ Part 3 Ex3App fixed ☐ lab rules (page 8) signed ☐ submitted

Date **Demonstrator's Initials**

Laboratory 2 Variables

Notes

Reading: L,D&C Chapter 1, Sections 1.1,1.2 (pages 2 - 18), Chapter 2, Sections 2.1 - 2.3 (pages 34 - 50).
You may feel that we have thrown you in at the deep end, but don't panic. Every concept introduced will be revisited in its own time. For now, we ask you to look for patterns in the code examples you are given in lectures, in the text and in this lab book. The lab work can be done by following the patterns. We don't expect you to understand it all at this stage.

Preparation

Preparation questions are (mostly) taken from L,D&C as indicated, and you will need to read the textbook to answer them. The questions below should be completed before you come to the lab session. Write your answers in the space provided.

1. Give examples of two types of Java comments and explain the differences between them.

2. Exercise EX 1.2 (L,D&C page 28). Which of the following are not valid Java identifiers? Why?

Identifier	Valid?	Reason
a. Factorial	☐	
b. anExtremelyLongIdentifier	☐	
c. 2ndLevel	☐	
d. level2	☐	
e. MAX_SIZE	☐	
f. highest$	☐	
g. hook&ladder	☐	

3. What is the name of the method where statement execution begins in a Java program?

4. Exercise EX 2.4 (L,D&C page 69). What output is produced by the following statement? Explain.

```
System.out.println("50 plus 25 is " + 50 + 25);
```

5. A Java program contains the following statements:

```
int i = 7;
int x = 3;
x = i;
i = x;
```

What will the value of `i` be after these 4 statements have been executed?

What will the value of `x` be after these 4 statements have been executed?

6. A Java program contains the following statements:

```
String first = "Easter Break";
String second = "University of Otago";
String day = "19";
String month = "04";
int year = 2019;

System.out.print(day + "/" + month + "/" + year);
System.out.println(first + "\n" + second );
System.out.println(year + 10);
System.out.print(month + 1);
```

What will the output look like?

Lab Work

The three parts to this lab are designed to give you experience working with variables. Note that we will use the javadoc form of block comment `/**` for class descriptions and method headers from this lab on. (L,D&C end of Section 12.4).

Part 1

1. Open a new **DrJava** file. Type in the following code:

```
/** Lab 2 COMP160 Part 1 TwoNumbersApp.java
 * (your name) January 2019
 */
public class TwoNumbersApp{
   public static void main (String[] args){
      double num1;   //declaration, specifies data type and name
      double num2;
      num1 = 8.5;   //assignment
      num2 = 15.0;
      System.out.println("First number is " + num1);
      System.out.println("Second number is " + num2);
      System.out.println("Sum is " + num1 + num2);
   }
}
```

> **DrJava has a very useful function for automatically indenting your code.** The tab key will indent any selected piece of code. Indent your whole class by choosing **Edit> Select All (⌘A)** then **tab**.

2. Save the file in your **Lab2** directory as `TwoNumbersApp.java`. Compile your code. If there are errors in your code, you will see them now in the Compiler Output window. If you typed carefully, you will again see the message "Compilation completed.". When you alter code, you need to compile the changes to make a new class file before you run it again.

3. Run your program. The program output will appear in the Console window. The Console window accumulates output from all programs run during a session. The Interactions window, however, is reset each time a program is run. It will be displaying the java instruction to run `TwoNumbersApp` and the program output.

The Console window can be refreshed using the instruction **Tools> Clear Console**.

> The sign you have known in mathematics as equals (=) needs in Java to be thought of as "gets the value" or "is assigned the value". It is known as the **assignment** operator.
>
> The statement `num1 = 8.0;` assigns the value `8.0` to the variable `num1`. The order is important! The value on the right of the assignment operator is assigned to the variable on its left.
>
> A variable must be **declared** (given a name and data type) before it can be used.
>
> The + sign can mean **concatenation** or **addition**. If either of the operands is a string (sequence of characters) rather than a number, the + operator will be unable to perform addition e.g. there is no sensible answer to `"cat" + 3`. In this case, concatenation is performed – the second operand is treated as a string and is appended to the first – producing for the example given `"cat3"`.

4. Check the output of your code. Does the answer look correct?

5. Fix your `TwoNumbersApp` program so that the numbers are **added** rather than **concatenated**.

6. Save, compile and run your code.

Part 2

1. Use **File> Save As** to save your `TwoNumbersApp` class as a new file called `ThreeNumbersApp.java` then change the class name in the code to match the new filename.

2. Open your **Home> COMP160 > Lab2** directory. Resize the open windows so you can see the `DrJava` window and the **Lab2** window at the same time. Watch the `Lab2` window as you compile your `ThreeNumbersApp` class for the first time. What is the name of the file that appears after the compile? _____

The next steps will show you how to alter the class `ThreeNumbersApp` so that it **stores** then displays the sum of **three** numbers.

3. Declare a third variable, of type `int`, called `num3`. Assign a value to `num3`.

4. Insert another `println` statement which describes and displays the value stored in `num3`.

5. Declare a fourth variable called `sum`. It is going to hold the sum of two `doubles` and an `int` so will need to be of the most precise data type involved (`double`).

6. Assign to the variable `sum` the sum of the three variables.

7. Alter the `"Sum is "` statement so that it prints the value stored in the variable `sum`.

Part 3

Copy the file called `FillRestaurant.java` that can be found in the **coursework> COMP160> coursefiles160> Lab02** directory to your **Lab2** directory. Open it in `DrJava`.

> You may notice that double-clicking on a `.java` file will not open the file in DrJava. Drag the file to the open DrJava window or use **File > Open**.

The class will not compile, nor produce any useful output, at this point.

```
/** COMP160 Lab2 SS 2019    FillRestaurant.java
 * A restaurant caters for large tour groups and takes bookings by the bus and van load each evening.
 * This class stores the booking data and calculates the number of unallocated seats.
 */
public class FillRestaurant{
    public static void main(String[]args){
        final int MAX_OCCUPANCY = 300;      // number of seats in restaurant
        final int BUS_CAPACITY = 35;        // number of seats in a bus
        final int VAN_CAPACITY = 8;         // number of seats in  a van
        final String DATE = "30th January 2019";  // dining date

        int numBusBooked =  4;              // number of buses expected on DATE
        int numVanBooked = 2;               // number of vans expected on DATE

       // number of diners expected from buses,   * is the multiplication sign
        int busDiners = numBusBooked * BUS_CAPACITY;
       // number of diners  expected from vans
        int vanDiners =
          . . .
```

1. Finish the statement which calculates and stores the number of diners expected to be arriving in vans.

2. Write another statement which calculates and displays the number of seats left in the restaurant for that date, producing the output:

```
Seats left for 30th January 2019 : 144
```

Use the DATE variable to produce the date in the output string rather than typing it.
When you have completed these tasks, call a demonstrator to mark your work and record your mark.

Lab 2 completed

The code I present for my Lab 2 mark is my own work according to the definition on page 7 _____

☐ preparation exercises ☐ Part 1 TwoNumbers ☐ Part 2 ThreeNumbers
☐ Part 3 FillRestaurant ☐ DATE used ☐ comments ☐ submitted

Date **Demonstrator's Initials**

Laboratory 3 Methods

Notes

Readings: L,D&C Chapter 2, Section 2.6 (pages 61-65).
Until now all your code has been in the main method. In this lab you will be writing several methods in one class, and sending data from one method to another.

You will need to look carefully at the code examples from your Lecture 3 notes in order to answer some of the preparation questions.

Preparation

1. **a.** The class `Rhyme` below defines three **methods**. What are their names?

b. The class `Rhyme` below has declared two **local variables**. What are their names?

c. The `main` method in the class `Rhyme` below contains two method calls. Number them in order of execution.

```
public class Rhyme{

    public static void main(String [] args){
        displayString2();
        displayString1();
    }// end main

    /** method which displays whatever is stored in a local string variable to the screen*/
    public static void displayString1(){
        String s1 = "Is this just fantasy?";
        System.out.println(s1);
    }//end displayString1

    /** method which displays whatever is stored in a local string variable to the screen*/
    public static void displayString2(){
        String s2 = "Is this the real life?";
        System.out.println(s2);
  }//end displayString2

}//end class
```

d. What will the output of this code be?

2. Since Lab1, you have called the method `println` without having to write a definition for it. How is this possible?

3. What output would the following code produce?

```
public class Lab3{
    private static int n;  // this variable is declared outside of any method, so is a data field

    public static void main(String[]args){
        methodThree();
        methodOne();
        methodTwo();
        methodOne();
        methodOne();
        System.out.println("n is " + n);
        methodThree();
        System.out.println("n is " + n);
    }

    public static void methodOne(){
        n = n * 2;
    }

    public static void methodTwo(){
        n = n + n + n;
    }
    public static void methodThree(){
        n = 2;
    }
}
```

Lab Work

A variable declared **within** a method is called a **local variable**, as it is only able to be used/accessed within that method. This is a safety feature – in different sections of a large program, two programmers in a team can each choose to use the same variable name in their method without affecting each other.

Because a variable declared within a method is local to that method, if its value is needed elsewhere in the program it must be sent (passed) as a parameter. The local variables in this program are in the main method.

Data fields are declared outside of any method, and are available to be used by every method. They are the data held by the class: its state. All data fields and methods in the application class (with the `main` method) must be `static`.

Part 1

1. Make a copy of the file `Letter.java` from the **coursework>COMP160>coursefiles160> `Lab03`** folder to your **`Lab3`** folder. Open it in **DrJava**. Identify the 2 methods, 2 data fields and 3 local variables.

```
/** Lab 3, COMP160, 2019 */

public class Letter{
    private static String yours = "Yours sincerely";
    private static String sign = "Mr Albert Agnew Esq.\nHuman Resources Manager\n
                                  Buttery Baps Unlimited\nwww.bb.co.nz";

     public static void main(String [] args){
         int junior = 25000;          // standard pay rate for Junior employee
         int intermediate = 35000;    // standard pay rate for Intermediate employee
         int senior = 50000;          // standard pay rate for Senior employee

     } // end method

      /** displays a job offer for Bottle Washer at $25K */
     public static void jobOffer(){
        System.out.println("Dear applicant\n
                    I wish to offer you the position of Bottle Washer.\n
                    The pay rate will be $25000 per annum.");
        System.out.println(yours + "\n" + sign + "\n");
     } // end method

} // end class
```

2. Compile and run. There will be no output. Why not?

3. Call the `jobOffer` method from the `main` method. Compile and run. The output will give you a clue to what the `\n` is doing. It is called an escape sequence. The character `n` following the character `\` will produce a new line. (Refer to L,D&C Listing 2.4 page 40)

This method can only ever be used to offer one job (Bottle Washer) and one pay rate ($25000).

4. Let's add some flexibility. Write a second `jobOffer` method in the `Letter` class.

```
public static void jobOffer (String jobTitle){
   System.out.println("Dear applicant\nI wish to offer you the position of " +
                        jobTitle + ".\nThe pay rate will be $25000 per annum.");
   System.out.println(yours + "\n" + sign + "\n");
} // end method
```

5. Compile your code. If you run your code now, you will see no change because you haven't called this new method yet. Call your new method from the `main` method, twice, each time sending it a different job title e.g. `jobOffer("Chief Cook");`

How does Java know which of the 2 `jobOffer` methods to run? It matches the order and data type of the actual parameters to the formal parameters. The method call `jobOffer()` has no parameters, so will invoke the method which expects none, namely **`public static void`** `jobOffer()`, while the method call

`jobOffer("Chief Cook")` has one actual parameter of type String, so will execute the method which expects one String, namely **`public static void`** `jobOffer(String jobTitle)`.

This is called method overloading. *(Java will not allow you to have 2 methods with the same "signature" (method name and number, type and order of parameters) in the same class. See lecture 23.)*

6. **Add further flexibility to your code by passing the pay rate as a parameter too.** Copy (and paste) the second `jobOffer` method, then adapt the copy to take a second parameter: an integer called `payRate`. The method should display this job title and pay rate, rather than the fixed 25000 of the previous method.

   Call this method twice from the main method, offering a different job title each time, and a different standard pay rate each time. There will be a compiler error if the order of your actual and formal parameters does not match.

7. **Personalise the message by sending the applicant's name as a parameter.** Copy (and paste) the third `jobOffer` method, then adapt the copy to take a String called `applicant` as an input parameter as well as the existing `payRate` integer and `jobTitle` String. The method should display the offer adressed to the name, but otherwise producing a similar output to the first and third `jobOffer` methods only this method is even more flexible. Test your method by calling it from the main method, e.g.

   ```
   jobOffer("Henry Hall", "Bean Counter", senior);
   ```

   Note: Java has no way of knowing which String was intended to be a job title and which was intended to be the name. It is the programmer's job to check the data is being sent in the order expected.

   On to the next challenge - user input! The method you have just written will receive data typed in by the user (you) while the program is running rather than hard-coded data as above.

8. Add the line **`import`** `java.util.Scanner;` to your code file, above the first comment and the class header. This makes the library class `Scanner` easier to use in your class.

9. Following the `Echo` example (L,D&C Listing 2.8 page 64), write code at the start of the `main` method which:
   - declares a local `String` variable called `name`
   - constructs a `Scanner` instance object using the line `Scanner scan = new Scanner(System.in)`
   - prompts the user with a meaningful message (the program will sit and wait for input - the user needs to be told what is expected of him/her) e.g. "Enter the successful applicant's name".
   - calls the `nextLine` method on the `Scanner` object to read text input from the user, and assigns it to the `name` variable.

   Write another call to the fourth `jobOffer` method using `name`, a job title, and one of the standard pay rates as actual parameters.

The most flexible and useful methods contain very little actual data. Their data comes in from the outside via the parameter lists, so the method's behaviours can be used repeatedly with different data.

10. It is possible, and common, to call methods from other methods. Write another `void` method called `signature`, and paste one of the `"yours . . . sign"` statements from the `jobOffer` methods into its body. Change all the `jobOffer` methods so they call this `signature` method to display the signature.

    (With this structure it is debatable whether the data fields are required any more – they could be local variables in the `signature` method. As local variables they would need to lose the `private` and `static` modifiers. They are private because they are local variables. They will be static because they are declared in a static method.)

11. Write comments in your code - there are some examples of commented code in **coursework> COMP160> coursefiles160> `Resources`**. Each method should have a preceding block comment describing its purpose e.g. */** displays a job offer */* .

Part 2

1. Take a look at the online documentation (link below). **Bookmark this page**, because you will be needing it again and again. The top left window lists Java packages (organised groups of classes for a particular purpose). Scroll to `java.util` and click.

The lower left window will change to give you a list of links relevant to `java.util`. Scroll down until you find `Scanner` and select it.

The main window will now show information relevant to the `Scanner` class. Scroll down until you find Method Summary, then further until you find `nextInt()` and `nextLine()`.

Record the brief description of the `nextInt()` method, and the return type of the method (it is in the column on the left).

_____

Click on the link for a fuller description of any particular method, but don't worry if it doesn't mean anything to you just yet.

2. Have a look at `java.lang` and from there the class `System`.

This is how you find out about the packages of classes in the libraries, and what they can do.

```
http://docs.oracle.com/javase/8/docs/api/index.html
```

`UML diagrams`

It can be useful to represent the class structure in a diagram. One form of diagram in common use is the Unified Modeling Language. The UML class diagram for the finished `Letter` class is shown below. It has three sections.

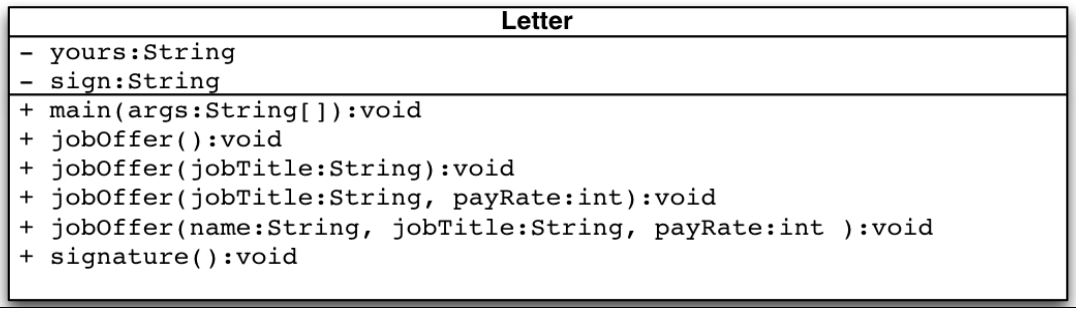
The top section is for the name of the class.

The middle section is for the data fields. Data fields are variables declared outside of any method. There are two String data fields. They are listed with their attributes in this order:

1. visibility **`-`** represents "private"
2. the name of the data field
3. a colon **`:`**
4. the data type of the data field (these data fields are both Strings**`)`**

The lower section lists the methods. The methods are listed with their attributes in this order:

1. visibility **`+`** represents "public"
2. the name of the method
3. parentheses
   may be empty **`()`** or contain a list of the parameter/s, separated by a comma **`,`** in the format
   `name:data type` *(the main method's parameter is an array of String values called args – you will cover arrays in the second half of the course)*
4. a colon **`:`**
5. the method's return type (these methods are all **`void)`**



Lab 3 Completed

The code I present for my Lab 3 mark is my own work according to the definition on page 7 _____

☐ preparation exercises ☐ Part 1 working ☐ 4 jobOffer methods ☐ comments
☐ signature method ☐ Part 2 API bookmarked ☐ submitted

Date **Demonstrator's Initials**

Laboratory 4 Expressions

Notes

Readings: Lecture 4 notes and L,D&C Chapter 2, Section 2.4 (pages 51 – 61).

You will be using `return` methods. There are many new concepts incorporated into this lab. Don't panic. Give yourself time and work through them one by one.

Preparation

1. Exercise EX 2.10 (L,D&C page 69). Given the following declarations, what result is stored in each of the listed assignment statements?

Remember: If both operands are integers the result will be an integer. Any fractional part is lost. If this result is then stored as a `double` or a `float` (as in `b.` or `d.` below), the integer will be automatically widened (displayed with `.0` after it).

```
int iResult, num1 = 25, num2 = 40, num3 = 17, num4 = 5;
double fResult, val1 = 17.0;
```

a. `iResult = num1 / num4;`

b. `fResult = num1 / num4;`

c. `iResult = num3 / num4;`

d. `fResult = num3 / num4;`

e. `fResult = val1 / num4;`

g. `iResult = num1 / num2;`

n. `iResult = num3 % num4;`

o. `iResult = num2 % num3;`

p. `iResult = num3 % num2;`

q. `iResult = num2 % num4;`

2. A `return` method can only ever return one single value.

The type of this value must be specified in the method header in place of the keyword `void`.

What data type is being returned by methods with the following headers (yes, it is as easy as it looks):

a. `public static int aMethod()` returns a value of type _____

b. `public static double bMethod()` returns a value of type _____

c. `public static String cMethod()` returns a value of type _____

d. `public static int dMethod(double d)` returns a value of type _____

e. `public static double eMethod(String s, int i)` returns a value of type _____

f. `public static String fMethod(String s)` returns a value of type _____

3. The result of a `return` method (the value returned) can be used in other parts of the program by simply calling the method. A method is called by its name and a parameter list (which may be empty). Imagine the value which is being returned replacing the method call in the flow of the program.

Legal method calls for the methods in Exercise 2 (above) could look like these (any legal parameter values may be used in the parameter list):

```
double sum = aMethod() + bMethod();   //adds the int returned by aMethod and the double
                                      //returned by bMethod
```

```
System.out.println("The result is " + dMethod(25.3)); // displays an int
```

Note: `dMethod` must be sent a parameter which is (or can be widened to) a `double` or the code will not compile. Widening refers to the automatic upgrade of an integer to a double when necessary e.g. an input of `35` widens to `35.0` if a double is expected.

```
System.out.println(cMethod());          //displays a String

System.out.println("The result is " + eMethod("xE0", 3));   //displays a double
```

Note: `eMethod` must be sent a `String` and an `int` input parameter, in that order, or the code will not compile.

a. Write a statement which stores in a variable called `difference` the value resulting from subtracting 2.3 from the value returned by `bMethod`. Include the declaration for the variable `difference`.

b. Complete the method definition for `fMethod`. The method should return the input string twice, separated by an asterisk e.g. if the input string is `"RUN"` the output would be `"RUN*RUN"`

```
public static String fMethod(String s){



}
```

c. Complete the statement which assigns to a variable called `result` the string returned by `fMethod` (as described above) , using the string `"RUN"` as the input (actual) parameter.

```
String result =
```

d. Complete the statement which assigns to a variable called `result` the string returned by `fMethod,` using the string returned by `cMethod` as the input (actual) parameter.

```
String result =
```

e. Write a complete method definition for `dMethod` from Ex .2 , which returns double the value of its input parameter, cast into an int e.g. the method call `dMethod(5.2)` would return `10`.

Lab Work

You are provided with an incomplete program to finish. When completed it will read, in turn, three Fahrenheit values from the user and convert them to Celsius.

You will find it helpful to refer initially to code listing 2.7 (L,D&C page 55) which performs a single Celsius to Fahrenheit conversion and was the starting point for the code provided. Where the code-listing in the text book is limited to converting a single Celsius temperature (24), the code for this lab will be more flexible by getting each Fahrenheit value from the user. The Fahrenheit input values will need to be read as `double` rather than `int`, so you will need to use the `Scanner` class's `nextDouble()` method rather than `nextInt()`.

1. Copy the file `FahrenheitToCelsius.java` to your **`Lab4`** directory and open it in **`DrJava`**.

```
import java.util.Scanner;
/** Lab 4  COMP160 SS 2019
 * Starting code
 */
public class FahrenheitToCelsius{
  public static void main(String[]args){
    convertFToC();
    //Step 5
  }

  /** gets input from user representing fahrenheit and displays celsius equivalent*/
  public static void convertFToC(){
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter Fahrenheit temperature");
    double fahrenheit =           ; //Step 2 - assign next double input from Scanner object
    System.out.println( fahrenheit + " degrees Fahrenheit is " ); //Step 4
  }


  /** calculates and returns the celsius equivalent of a double input parameter called fahr*/
  public static double toCelsius(double fahr){
    final int BASE = 32;
    final double CONVERSION_FACTOR = 9.0/ 5.0;
    //Step 3
    return celsius;
  }
}
```

2. In the `convertFToC` method, assign to the variable `fahrenheit` the next `double` input from the keyboard.

3. In the `toCelsius` method, write an expression which calculates the celsius equivalent of the variable `fahr` and stores it in a variable called `celsius`.

You may like to determine the formula required by working your way from the Celsius to Fahrenheit conversion formula in the space below. Try to isolate celsius on one side of the equation. Maintain equality by performing equal operations on both sides. Call a demonstrator if you need a hand.

```
fahr = celsius * CONVERSION_FACTOR + BASE
```

4. In the `convertFToC` method, change the second `System.out.println` statement so it produces a display such as

```
212 degrees Fahrenheit is 100.0 degrees Celsius
```

for an input of 212. If you are baffled by this instruction, here is a fuller description:

The first half of the output line (212 degrees Fahrenheit is) is already written. The next item to be displayed is the number of degrees celsius, which is being calculated and returned by the `toCelsius` method.

Remember (see Lecture 3 and preparation exercises) that the result of a **`return`** method can be used elsewhere in a program by calling its name and providing the correct number and type of input parameters.

Call the `toCelsius` method within the `System.out.println` statement, sending it the data that it needs (the variable representing the degrees in fahrenheit).

Finish off the statement by concatenating the string " degrees Celsius" on the end.

This program should now compile. If you run it, it will convert one value from Fahrenheit to Celsius.

5. Add two statements to the `main` method so the program will convert 3 fahrenheit values when it is run.

6. Run your program, and use it to calculate the celsius temperatures for the examples below. (In lecture 8 you will learn how to limit the number of decimal places displayed. Don't worry about it for now.)

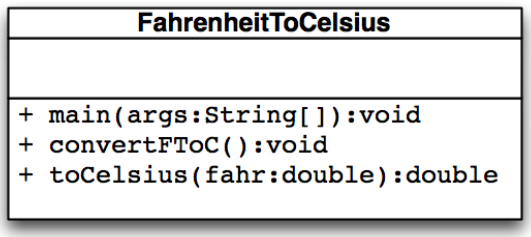
32.0 degrees Fahrenheit is degrees Celsius

98.6 degrees Fahrenheit is degrees Celsius

212.0 degrees Fahrenheit is degrees Celsius

UML class diagram

The UML class diagram for the FahrenheitToCelsius class shows one method with a return type other than `void`. The `toCelsius` method takes one `double` called `fahr` as an input parameter and also returns a `double`.



Lab 4 Completed

The code I present for my Lab 4 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments

☐ temperature converter working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 5 Graphics

Notes

Readings: L,D&C Graphics Appendix F (pages 965 – 973).

This lab marks a significant change in the design of our programs. From now on, most of our programs will have a more usual Object Oriented design. As well as the application class we will create and use one or more instances of support classes.

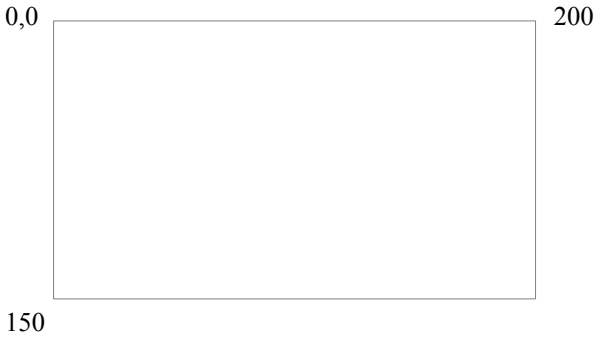
In Lab 3 you created an instance of the class `Scanner` using the keyword `new`. In Part 1 of this lab, you will be working with an instance of the class `SnowmanPanel`, which is created by the keyword `new` in the `main` method of the `SnowmanApp` class. The `SnowmanPanel` class uses the `Graphics` class from the package `java.awt` to "draw" shapes on the panel (JPanel). Notice that the methods in `SnowmanPanel` are not static.

Preparation

1. Exercise EX F.3 (L,D&C page 982). Assuming you have a `Graphics` object called `page`, write a statement that will draw a line from point `(20, 30)` to point `(50, 60)`.

2. Exercise EX F.4 (L,D&C page 982). Assuming you have a `Graphics` object called `page`, write a statement that will draw a rectangle outline with height `70` and width `35`, such that its upper-left corner is at point `(10, 15)`.

3. In the diagram of a graphics window below, sketch a circle **centred** on point `(50, 50)` with a **radius** of `20` pixels. Draw the circle's bounding box, and mark where the left and top co-ordinates intersect with the axes.



4. Exercise EX F.5 (L,D&C page 982). Assuming you have a `Graphics` object called `page`, write a statement that will draw a solid (filled) circle *centred* on point `(50, 50)` with a radius of `20` pixels. This is the circle you sketched in Exercise 3 above.

5. What is the purpose of the `TOP` and `MID` local variables in the `SnowmanPanel` class listed on pages 971 – 972 of L,D&C?

Part 1

Adapted from Programming Projects PP F.1 (L,D&C page 982). Create a revised version of the `SnowmanPanel` program as described below.

You will find the code in the **`coursefiles160`** folder. The `main` method is complicated. You don't need to understand what is happening in the `main` method until much later in the course.

1. Copy the `SnowmanPanel.java` and the `SnowmanApp.java` files to your **`Lab5`** folder. Open them in **DrJava**.

The `main` method creates an instance of the `SnowmanPanel` class using the keyword `new`. The `SnowmanPanel` class contains the `paintComponent` method, which is called automatically. It is the code within the `paintComponent` method that we want you to concentrate on at present. Notice how useful the comments are to your understanding.

Compile the files (in DrJava all open files are compiled when you select Compile). **Run the `SnowmanApp` class.** You should see a snowman similar to that pictured on L,D&C Appendix F Listing F.1 /F.2 page 971.

2. Add two red buttons to the upper torso. Make sure they are centred on the MID line.

3. Make the snowman frown instead of smile. (See "Drawing an arc" below)

4. Move the sun to the upper-right corner of the picture.

5. Display your name in the upper-left corner of the picture.

- **Note** that a string is drawn above and to the right of its starting point

6. Shift the snowman to the left of the picture. (**Hint:** This requires just one statement to be altered.)

Drawing an arc (smile / frown)

```
page.drawArc (MID-10, TOP+20, 20, 10, 190, 160);  //arc which draws the Snowman's smile
```

The Snowman's smile is an arc. The first 4 numbers define the bounding box of the oval of which the arc is a portion. In the diagram below (Figure 5.1), the bounding box is outlined in gray, and the complete oval has been drawn in outline.

The fifth number is the start position of the arc. Position `0` is at 3 o'clock, and the measure is degrees (there are 360 degrees in a circle). The direction is anti-clockwise. The Snowman's smile starts at `190` degrees.

The sixth and last number is the **size** of the arc angle. The Snowman's smile is a 160 degree angle, taking the arc from 190° to 190 + 160 = 350° (almost back to `0`). In `SnowmanPanel.java`, because `drawArc` rather than `fillArc` is used, just the edge of the specified arc is drawn. In Figure 5.1, the whole of the specified arc is filled in as though `fillArc` has been used because it is easier to visualise the size and shape of the arc with `fillArc`.

Diagram of the arc for the Snowman's smile.

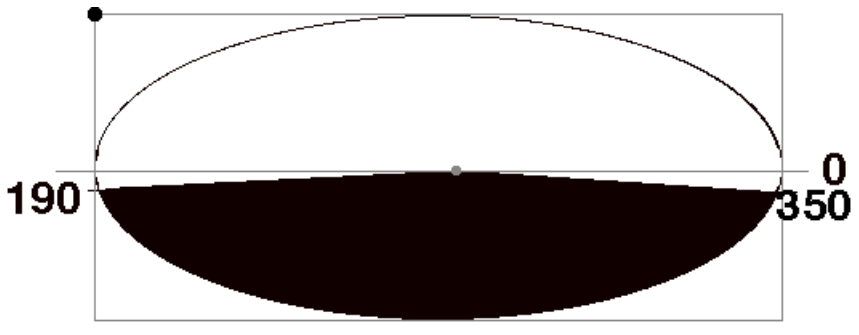


Figure 5.1

Part 2

Adapted from Programming Project F.7 (page 983). Write an application that shows a pie chart with eight equal slices, all coloured differently. Lastly, move one piece of the pie out a little as shown below in Figure 5.2.

We have provided you with the class structure required. Copy `Pie.java` and `PieApp.java` from **LabFiles** to your **Lab5> Part2** folder. Compile the files and run the application class. You now have a blank slate on which to draw your pie chart.

L,D&C lists colour names in Appendix F Figure F.2 page 967, and describes how to draw an arc on page 969. Figure F.5 page 969 will be a useful reference. (Your arcs will need to be filled.)

FIRST, plan your arcs by filling in the table below. There is a table column for each specification required in order to draw an arc. A full circle is 360º (its `arcAngle` would equal 360).

	x location	y location	width	height	startAngle	arcAngle
1st arc						
2nd arc						
3rd arc						
4th arc						
5th arc						
6th arc						
7th arc						
8th arc						

You can see from the table you have filled in that many of the values required are the same for each arc. **These values should be stored in local variables** with sensible names (similar to the way that MID and TOP were stored in the Snowman class). Your code can then be written very quickly, using the copy and paste commands for efficiency.

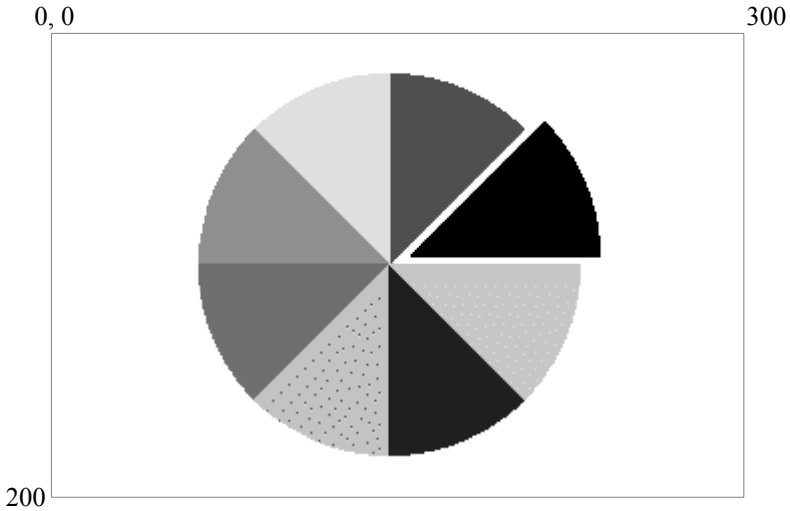


Figure 5.2

Lab 5 Completed

The code I present for my Lab 5 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments

☐ Part1 Snowman ☐ Part2 Pie (using local variables) ☐ submitted

Date **Demonstrator's Initials**

Laboratory 6 Objects

Notes

Readings: L,D&C Chapter 3, Section 3.1 (pages 76 – 80).

You have already created some objects, such as `Scanner` objects in Labs 3 and 4, `SnowmanPanel` and `Pie` objects in Lab 5. You have also created `String` objects in Lab 3, using the Java shortcut which doesn't require the keyword `new`. So creating / instantiating an object is really not too scary!

Once an object has been instantiated, its public methods can be accessed from other classes using dot notation. For instance, if the object has been given the name `book1`, its method `displayBook` can be accessed using the expression `book1.displayBook()`. This lab has you creating instances of a class, and manipulating them from an application class.

Preparation

1. Assume there is a class called `MyClass` which contains a `void` method called `display`.

a. What will the following statement in an application class do?

```
MyClass x = new MyClass();
```

b. Write a statement to follow the one above, which will call the `display` method on `x`.

c. Write a statement which will create an instance of `MyClass` called `y`.

2. Look at the listing for `Book.java` on page 34 of this lab book.

What are the names of the three data fields?

Which three of the seven methods are accessor methods?

What two keywords do the three mutator methods all have in their headers?

3. What is the difference between a data field and a local variable?

4. Which of these methods are accessor methods? Which are mutator methods? Which are neither?

```
public class Person{
    private int age;
    private String name, country_of_origin;

    public void methodOne(int ageIn){
        age = ageIn;
    }
   public void methodTwo(String nameIn, String originIn){
       name = nameIn;
       country_of_origin = originIn;
    }
    public int methodThree(){
        return age;
    }
    public void methodFour(){
        System.out.println(name + " " + country_of_origin);
    }
    public String methodFive(){
        return country_of_origin;
    }
    public void methodSix(){
        int birthyearApprox = 2019 – age;
    }
}
```

5. An application class intended to be used with the `Person` class is listed below.

```
public class PersonApp{

  public static void main(String[]args){

      Person person1 = new Person();

      person1.methodOne(19);

      person1.methodTwo("Brian Berry", "Ireland");

      System.out.println("Age is " + methodThree());  // line a. syntax error? Yes / No

      System.out.println(person1.methodFour());       // line b. syntax error? Yes / No

      System.out.println(person1.methodFive());       // line c. syntax error? Yes / No

  }

}
```

Two of the `System.out.println statements` will cause compiler errors. Identify them, and describe the errors.

1.

2.

Lab Work

Part 1

DrJava has an Interactions window which allows you to test bits of code straight away. You can type a Java statement directly in to this window and see its effect. You can also type a statement which uses the result of a previous statement. In this lab, you will create some objects, first using DrJava's Interactions window, then using an application class.

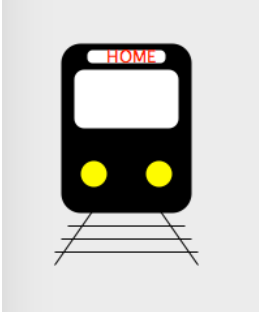
1. Copy the file `MyPanel.java` from the **coursefiles160** folder to your **Lab6** folder. Open it in **DrJava**. You can see a listing of the code on the next page. **Compile the code.**

2. Click on the `Interactions` tab. Underneath **Welcome to Java . . .** , type

   ```
   MyPanel mp1 = new MyPanel();
   ```

   then press <return>. You have just created an object - an instance of the class `MyPanel` - which you can now refer to by the name `mp1`.

   You will see this instance drawn at the top left of your screen. The default position is (0, 0) and the default background colour is gray.



3. Now type `mp1.decorate(java.awt.Color.blue, 180);` in the interactions window then press <return>. The frame has now moved and the background colour is blue.

   You can call the decorate method as many times as you like. (**Hint:** If you want to repeat or modify a previous instruction in the interactions pane, use the up arrow ↑ to scroll back through the previous instructions, then press <return> when the one you want is selected.)

   It is usually more convenient to manage your objects using an application class. All the required code can be conveniently gathered into a main method, which can be run with a single command.

4. Open a new **DrJava** document and in it write an application class for `MyPanel` called `MyPanelApp`. (By definition, the application class is the class with the `main` method in it.) The `main` method should contain the two instructions you have typed in the interactions pane, in the following order:

   ```
   MyPanel mp1 = new MyPanel();
   mp1.decorate(java.awt.Color.blue, 180);
   ```

5. Save this class as `MyPanelApp.java` in the same folder as your `MyPanel` class. **Compile** and run. You have now used an application class to create an instance of your `MyPanel` class. This instance is called mp1, and this name is used to call its methods.

6. In the `main` method, create another instance of `MyPanel`. Set its colour and x location by calling its decorate method. The `decorate` method is expecting to be passed a `Color` and an `int` in that order, separated by a comma. If it doesn't get what it wants, it will not compile.

   See L,D&C Appendix F Fig. F.2 page 967 for a list of Color names, or look up the `java.awt Color` class in the API.

   Compile and run. You should now have two instances of the `MyPanel` class, located in different places, and coloured differently.

Rather than use the full name e.g. `java.awt.Color.blue` in the call to the `decorate` method you could use the statement

```
import java.awt.Color;
```

at the top of the application class in order for the `MyPanelApp` class to be able to find the `Color` class.

Then

```
mp1.decorate(Color.blue, 180);
```

would be sufficient.

In the `MyPanel` class, the statement `import java.awt.*;` saves us from having to use the full names for `java.awt.Color`, `java.awt.Dimension` and `java.awt.Graphics.` These are all classes in the `java.awt` package.

```
import javax.swing.*;
import java.awt.*;
/**
 * MyPanel.java
 * Lab 6, COMP160 2019
 * Make our own version of a JPanel class
 */
public class MyPanel extends JPanel {
    private int SIZE = 300;
     /** Constructor for MyPanel. Don't worry for now how this method works.*/
    public MyPanel(){
       setPreferredSize(new Dimension(SIZE,SIZE));   // set size of JPanel
       JFrame frame = new JFrame();                  // create JFrame object
       frame.getContentPane().add(this); // add JPanel object to JFrame object
       frame.pack(); // wrap ("pack") JFrame object around JPanel object
       frame.setVisible(true);                       // set to visible
    }
   /** method sets the background colour and x axis location of frame. Once again don't
       worry how it works, just know how to call it */
    public void decorate (Color shade, int xPos) {
       // set background colour
       setBackground(shade);
       // get JFrame object and set x-position, y-position is 0
       getRootPane().getParent().setLocation(xPos, 0);
    }

  /** Make our own paintComponent method */
   public void paintComponent(Graphics g) {
     super.paintComponent(g);              // draws the basic JPanel
     int MID = 100;
     g.fillRoundRect(MID-50,50,100,130,30,30); // train body in default colour Color.black
     g.drawLine(MID-55, 220, MID-20, 170); // left track
     g.drawLine(MID+55, 220, MID+20, 170); // right track
     g.drawLine(MID-55, 210, MID+55, 210); //  track sleeper
     g.drawLine(MID-50, 200, MID+50, 200); //  track sleeper
     g.drawLine(MID-45, 190, MID+45, 190); //  track sleeper

     g.setColor(Color.white);   // change colour
     g.fillRoundRect(MID-30, 55, 40, 10, 8, 8); // destination screen
     g.fillRoundRect(MID-40, 70, 80, 45, 15, 15); // front screen

     g.setColor(Color.yellow);
     g.fillOval(MID-35, 140, 20, 20); // left light
     g.fillOval(MID+15, 140, 20, 20);   // right light

     g.setColor(Color.red);
     g.drawString("HOME", MID-15, 65); // destination  text

   }//end paintComponent

}//end class
```

The class `MyPanel` does not contain or need to contain a `main` method. It is a support class. The class will be created (instantiated) by another class, in this case, the application class (which **does** have a main method).

Part 2

1. Copy the file `Book.java` from the **coursefiles160** folder to your Lab6 folder. Open it in **DrJava**. You will find a code listing on the next page.

The major difference between the structure of `Book` and the structure of `MyPanel` is that the `Book` class does not use graphics - its text output will show in the Console window.

`Book` has 3 data fields. They are `private`, so are not accessible from other classes. The only way another class can see what is in them is to call the public methods of `Book` which return their values (the `"get"` methods often called accessor methods) or which display their values (the `displayBook` method).

The only way another class can change the values in the data fields is to use the three public `"set"` methods (mutator methods)

2. Write an application class for `Book` called `BookShopApp`, which creates 3 instances of `Book` using the default constructor. When you have created each `Book` object, use its `set` (mutator) methods to set the values of its data fields (don't bother with user input, just type the values as literals in the code itself).

The first book is called "Life of Pi", has 348 pages and sells for $28.90.

The second book is called "Mister Pip", has 240 pages, and costs $22.70.

The third book is of your own choosing.

3. When the data fields are set, call each book's `displayBook` method.

What would happen if you called the `displayBook` method before **the data fields are set? Try it if you are not sure.**

Your output should look like this:

```
The name of the book is Life of Pi
It has 348 pages.
You can buy this book for $28.90
*********************************
The name of the book is Mister Pip
It has 240 pages.
You can buy this book for $22.70
*********************************
The name of the book is ...
.
.
.
```

The **state** (values stored in the data fields) of these 3 instances is different, but their **behaviour** (the methods that can be called on them) is the same (see Readings at the back of this book, page 117).

Extension Exercise: Write another class for your program called DVD which holds, changes and displays relevant data for a DVD (e.g. name, zone, rating, run time). Get your application class to create some instances of the DVD class, and display the data. You will be duplicating some behaviours from the book class. Later in the course you will learn how related classes in a hierarchy can share behaviours which are common to them.

Lab 6 Completed

The code I present for my Lab 6 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ Part 1 MyPanelApp

☐ Part 2 BookShopApp ☐ submitted

Date **Demonstrator's Initials**

```
import java.text.NumberFormat; // for formatting price to 2 decimal places. See L,D&C  p.94 Listing 3.4
/**
 * Lab 6, Part 2, COMP160  2019
 * Book.java
 * Stores and displays information about an individual Book.
 */
public class Book{

   //data field declarations
   private String title;                    // book title
   private int numPages;                    // number of pages in book
   private double price;                    // retail price of book

    /**sets the value of the data field title to input parameter value
     * @param t title of book */
   public void setTitle(String t){
      title = t;
   } //end method

    /**sets the value of the data field numPages to input parameter value
     * @param n number of pages */
   public void setPages(int n){
      numPages = n;
   } //end method

    /**sets the value of the data field price to input parameter value
     * @param p the price */
   public void setPrice(double p){
      price = p;
   } //end method

    /**returns the value of the data field title
     * @return the title of the book */
   public String getTitle(){
      return title;
   } //end method

    /**returns the value of the data field numPages
     * @return the number of pages in the book */
   public int getNumPages(){
      return numPages;
   } //end method

    /**returns the value of the data field price
     * @return the price of the book */
   public double getPrice(){
      return price;
   } //end method

   /** displays formatted Book information to the console window */
   public void displayBook(){
      NumberFormat fmt = NumberFormat.getCurrencyInstance(); //for formatting price
      System.out.println("The name of the book is " + title);
      System.out.println("It has " + numPages + " pages.");
      System.out.println("You can buy this book for " + fmt.format(price));
      System.out.println("*********************************");
   } //end method

} //end class
```

UML class diagram

Often the application class will not contain data fields, as it is more concerned with managing the instantiation of other classes. Support classes will usually contain data fields.

The middle section of the UML class diagram is for the list of data fields. As described in lab 3, the order is:

1. visibility **–** represents "private"
(+ represents "public")

2. the name of the data field

3. a colon **:**

4. the data type of the data field

The lower section lists the methods. As described in lab 3, the order is:

1. visibility **+** represents “public”
(**–** represents “private”)

2. the name of the method

3. parentheses

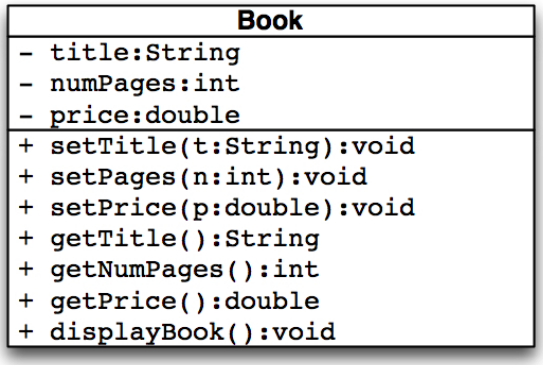
may be empty **()**

or contain the parameter/s in the format **`name:data type`**

if there is more than one parameter, there should be a comma between each e.g. `(n:int, p:double)`

4. a colon **:**

5. the method's return type (may be `void`)



Laboratory 7 Constructors

Notes

Readings: L,D&C Chapter 5, Section 5.1, 5.2, 5.3, 5.4 (pages 170 – 199).

- Constructors are special methods that have the same name as the class.
- Like other methods, a constructor can take input parameters.
- Unlike other methods, a constructor is neither `void` nor does it `return` a result.
- A constructor is only called on instantiation (creation) of the object.
- A class may contain one or more constructors – each will have a different parameter list.

The default constructor

Every support class is assumed to have a **default** constructor, with no parameters, that creates the object and initialises all the data fields to zero values. This default constructor doesn't have to be written.

To instantiate the class, the constructor is called from another class, often the application class, in conjunction with the keyword `new`:

```
Book book1 = new Book();   // the default constructor method call is underlined
```

The constructor with parameter list

There are several ways of setting the values of the data fields of an object. One is to specify values literally when writing the class, e.g. `private String title = "Jaws";` *// every Book object will have the title Jaws*

Another is to use a mutator method

```
public void setName(String titleIn){
    title = titleIn;   // title is set to the value of the input parameter
}
```

Other ways are to read in values typed by the user or read from a file.

Often the best way to set them is to assign values to the data fields **as the object is created** using a **constructor**. This

* makes sure that the data fields of the object are set to useful default values
* helps to avoid the problem of methods working with null data fields.

We could decide to define our own constructor that does something more than just create the object, for example:

✘
```
public Book(){ // this constructor must be in a class called Book, saved in a file called Book.java)
   title = "Jaws";  // sets the initial value of the data field title to Jaws (there is no capacity for other titles)
}
```

This constructor would set the `title` data field to *Jaws*. The problem is it would do so for every instance of the `Book` class. But every book is not called *Jaws*! A better way is to define a constructor that takes one or more input parameters and uses it/them to set the initial value/s of the data field/s. The actual data (in this case *Jaws*) is sent to the constructor from outside the class, allowing each object to have its own unique state on creation.

✔
```
public Book(String titleIn){  // note the generic variable name titleIn to receive the data
   title = titleIn;   //sets the initial value of the data field title to the value of the input parameter
}                     // note the similarities to the mutator method setName above
```

To instantiate the class, the constructor is called from perhaps the application class, using the keyword `new`:

```
Book book4 = new Book("Jaws");  // the constructor call, which sends the actual data, is underlined
```

Note that once you have written a constructor, you can no longer use the default constructor (`new Book()`)without specifically writing a replacement for the default constructor back in to your class. It would look like this:

```
public Book() {} // a replacement for the default constructor
```

With the replacement for the default constructor in the support (Book) class, even though other constructors are also present, the application class may again contain the default constructor call `new Book()`.

The first declaration statement below creates an instance of the class `Book` using the keyword `new` and the constructor call. It is stored in a variable called `book5`. The second stores the integer value `3` in a variable called `x`.

```
Book book5 = new Book("The Grapes of Wrath");
     int x = 3;
```

`int` is the data type of the variable called `x`. It is a **primitive** type.
`Book` is the data type of the variable called `book5`. It is a **reference** type.

Preparation

1. Write a constructor for the `Book` class from lab 6 which takes 3 parameters and uses them to set the values of all 3 data fields.

2. Write a statement (for the BookShopApp main method) which makes an instance of the Book class using the constructor you have written in Q1.

3. Which of the following could be legal constructor headers for a class called MyClass?

```
public int MyClass(){}
public MyClass(){}
public MyClass{}
public myClass(int x){}
public MyClass(int x, y){}
public MyClass(x:int){}
```

```
public int MyClass(int y){}
public MyClass(y){}
public class MyClass(String s){}
public void MyClass(String s){}
public MyClass(int 1st){}
public MyClass(int s, String x){}
```

4. A class `Exam` has an `int` data field called `score`. Write a constructor for `Exam` which sets the value of `score` to the value of its input parameter.

```
public class Exam{
  private int score;
```

}*//end class Exam*

5. Write a single statement in the `main` method of the application class below. It should create an instance of `Exam` called `exam1` by calling the constructor you wrote in the previous exercise. Send the constructor the data required to set its data field `score` to `80`.

```
public class ExamApp{
  public static void main(String [] args){


  }
}
```

6. A class `Hotel` has a `name` data field and a `num_beds` data field. Write a definition for the class `Hotel`. Include a constructor which sets the value of `name` and `num_beds` to the value of its input parameters. Also include a mutator which sets the value of `num_beds` to the value of its input parameter.

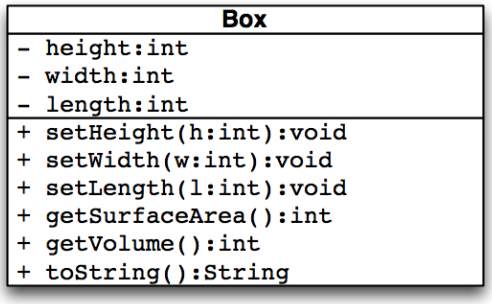
7. Write a `toString` method for the `Hotel` class which returns a String describing the hotel's name and number of beds as stored in the data fields. (A `toString` method is a very particular sort of method which **must** return a String, and takes **no** parameters).

8. Why is it useful to be able to pass parameters / arguments to constructors?

9. Both mutators and constructors enable you to store data in an object. How are they different?

Lab Work

This lab will guide you through the creation of a support class called `Box` which calculates the volume and surface area of a box given its height, length and width. An application class called `BoxApp` will create instances of `Box` using the constructor methods of `Box`. Refer to the UML class diagrams for an overview of the class structure required. Finally your support class will have a static data field, which is shared by all instances.



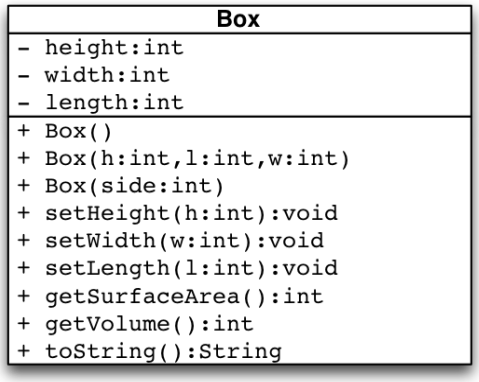
1. Write a support class `Box` which has three data fields as described in the UML class diagram.

2. Write a mutator method for each of the data fields. Mutators are methods which set the value of the data field to the value of the input parameter. They usually have names beginning with `set` e.g. `setHeight`

3. `Box` will need (accessor) methods which compute and return the volume and surface area. Note that volume and surface area are **not** stored as data fields. The volume is defined by the product of the height, width, and length. The surface area can be calculated by summing the areas of the six sides. (Hint: There are actually only three unique sides).

4. Write a method called `toString()` that will return a string describing the height, length, width, volume and surface area in the Console (see step 11).

The `Box` class now requires three different constructors, as described in the second UML diagram.



5. Write a constructor that takes three parameters, and uses them to set the `height`, `length` and `width` data fields.

6. Write a replacement for the default constructor that creates a `Box`. (All dimensions will be set to the default value of `0`.)

7. One possible variety of `Box` is a cube, where `height, length` and `width` are all the same. Write a third constructor for the class `Box` which takes just one input parameter and sets all the data fields to this value.

8. Create a new class called `BoxApp`. This will be the application class. In the `main` method, create a `Box` object (`box1`) using the replacement for the default constructor.

```
Box box1 = new Box();
```

9. Write a statement to display the String returned by this object's `toString` method.

10. Compile and run your code. You should see something like the line below, because the data fields are all set to 0 (the default value).

```
Height is: 0, Length is: 0, Width is: 0, Volume is: 0, Surface Area: 0
```

11. Call the mutators on the `box1` object to set the data fields to height 4, length 4 and width 6 **before** you call the `toString` method but **after** you have created the object. (You could set the values by user input with Scanner if you wish.)

12. Compile and run. Now the data fields have values, so the output should look something like:

```
Height is: 4, Length is: 4, Width is: 6, Volume is: 96, Surface Area: 128
```

13. There is an easier way to create an instance of `Box` and fill its data fields with values. Have the `main` method create a `Box` object (`box2`) with height set to 3, length set to 4 and width set to 5 by calling the constructor you wrote in Step 5.

```
Box box2 = new Box(3, 4, 5);
```

14. Write a statement to display what is returned by this object's `toString` method.

Note: Which constructor is called depends on the number of input parameters. When you created `box1` you called the constructor with no parameters and the constructor that accepts no parameters is run. When you created `box2` you called the constructor with 3 parameters, so the constructor that accepts 3 parameters is run. This is called method **overloading**.

You now have two objects / instances of `Box` that have different states.

15. When you run your code, you should now generate (for the input values shown) the following output:

```
Height is: 4, Length is: 4, Width is: 6, Volume is: 96, Surface Area: 128
Height is: 3, Length is: 4, Width is: 5, Volume is: 60, Surface Area: 94
```

16. In the application class, make two more instances of `Box` using the cube constructor from Step 7, setting the sides to `5` for the first and `7` for the second. Once again, display the description returned by each `Box's toString` method.

```
Height is: 4, Length is: 4, Width is: 6, Volume is: 96, Surface Area: 128
Height is: 3, Length is: 4, Width is: 5, Volume is: 60, Surface Area: 94
Height is: 5, Length is: 5, Width is: 5, Volume is:125, Surface Area: 150
Height is: 7, Length is: 7, Width is: 7, Volume is:343, Surface Area: 294
```

Part 2 Static

Static data is shared by all instances of a class. Static methods (including `main`) can only directly call other static methods.

Imagine that your `Box` objects are part of the cargo in a shipping container. Each box needs to be identified by its owner. There is just one owner for all the boxes in that container. It is possible that the owner changes in transit as part of a commercial transaction. A static data field can store this information.

17. In the `Box` class, add a `static String` data field to store the owner's name. Write a `static void` mutator method to set the owner to the value of its input parameter, and an accessor for the owner.

Adapt the `toString` method so that it also reports "`owned by`" and the owner's name.

18. At the **beginning** of the main method of the `BoxApp` class, set the owner to e.g. "Anna Austin". (Refer to Lecture 7 for the correct way to access class (static) members. Test that your code works, and all your boxes have an owner.

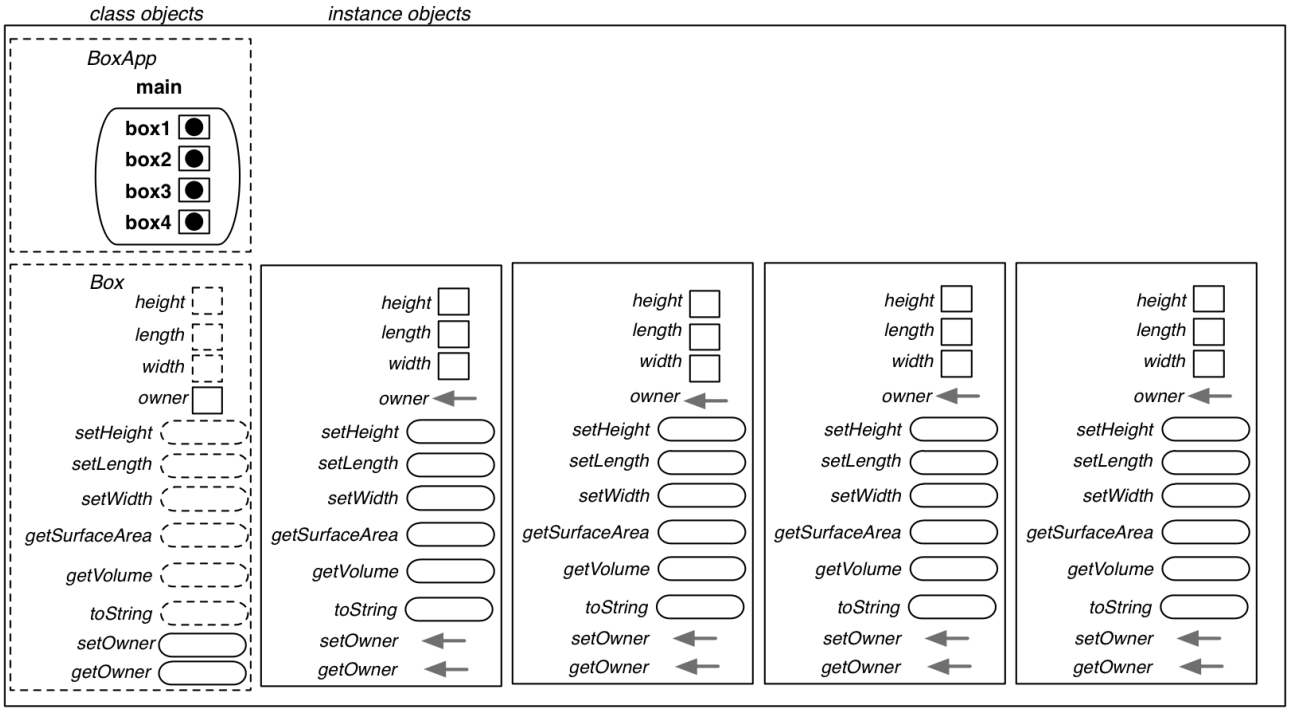
19. At the **end** of the main method of the `BoxApp` class, change the owner to e.g. "Bob Berry" and call all the `toString` methods again. Run the code to check that the owner has changed.

If you change the value stored in a class data field, it changes for all instances of that class. They are all sharing the value. For the `Box` class, all instances of `Box` must have the same owner at any moment in time.

Part 3 Self-documenting code with Javadoc comments.

20. In DrJava, open the Book.java file from Lab 6. With the Book.java window window active, select Tools > Javadoc > Preview javadoc for current document. Wait a few seconds. You will see an API for the class Book. This is possible because of the javadoc (**) comments above the class header and each method header.

Now return to your Box.java file. Write suitable javadoc comments in your Box.java class and create the Java documentation for this class.



Part 4 Class and Instance Object Model

21. Use the numbers from your lab work to complete the object diagram above by drawing arrows from the box1, box2, box3 and box4 references to the corresponding instance objects. Write the values which were requested by the Lab Work instructions in the **data fields** of each instance object.

Note how the static (class) members of the support class Box are presented differently from the instance members.

Lab 7 Completed

The code I present for my Lab 7 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ three constructors ☐ return methods - volume, surface area
☐ static datafield and methods ☐ javadoc and diagram ☐ working and submitted

Date **Demonstrator's Initials**

Laboratory 8 Math and Random

Notes

Readings: L,D&C Chapter 3, Sections 3.4 - 3.6 (pages 86 – 97).
The Java libraries contain many useful classes. Some of these classes contain methods for performing commonly required specific tasks such as formatting output, performing mathematical calculations and producing a random number. We will explore the use of some of these classes in this lab.

Preparation

1. Write a statement which creates a `Random` object called `generator`.

2. Adapted from Exercise EX 3.6 (L,D&C page 105). Assuming that a `Random` object called `generator` has been created, what is the range of the result of each of the following expressions? The first two are done for you.

a. `generator.nextInt(20)` 0 -19

b. `generator.nextInt(8) + 3` returns an int between 0 and 7 with an 'offset' of 3, giving 3 - 10

c. `generator.nextInt(50)`

d. `generator.nextInt(5) + 1`

e. `generator.nextInt(45) + 10`

f. `generator.nextInt(100) – 50`

3. Evaluate these expressions and indicate the data type of the result.

	expression	evaluates to	data type of result
a.	`Math.pow(3, 3) =`		
b.	`Math.pow(2, 4) =`		
c.	`(int) –5.8 * Math.sqrt(9) =` this is called casting – see lecture 4		
d.	`Math.floor(3.2) =`		
e.	`Math.floor(22.1) * Math.sqrt(9) =`		
f.	`Math.ceil(3.2) =`		
g.	`Math.round(3.2) =`		

There are 2 `round` methods in the `Math` class. Their short definitions are:

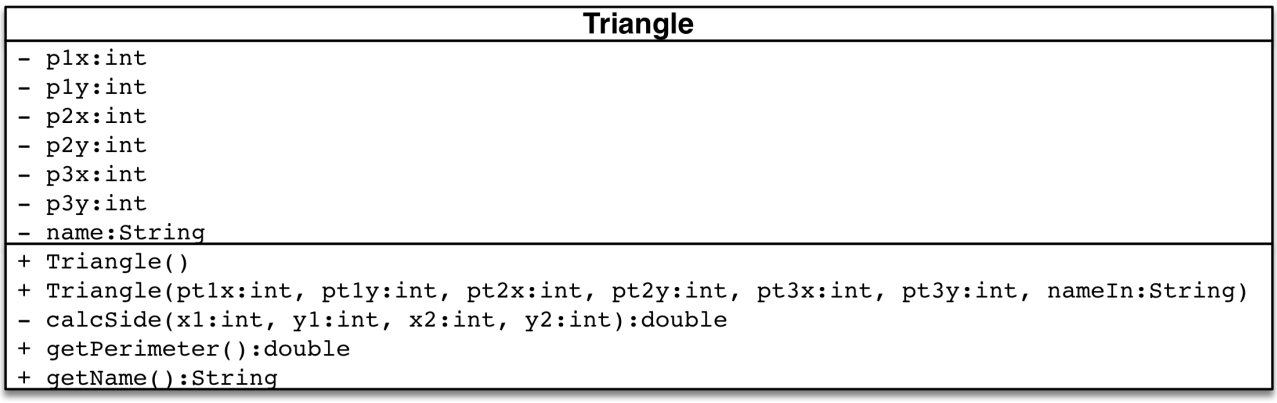
```
static long  round(double a)      Returns the closest long to the argument.
static int   round(float a)       Returns the closest int to the argument.
```

The method that is used depends on the data which is sent as a parameter. A floating point number (e.g. 3.2) is interpreted as a `double` unless it has F or f after it (e.g. 3.2F) in which case it is a `float`.
In **g.** above, the parameter is a `double`, so the first `round` method listed above is called.

4. Exercise EX 3.10 (L,D&C page 106). Write code statements to create a `DecimalFormat` object that will limit a value to 4 decimal places. Then write a statement that uses that object to print the value of a variable named `result`, properly formatted.

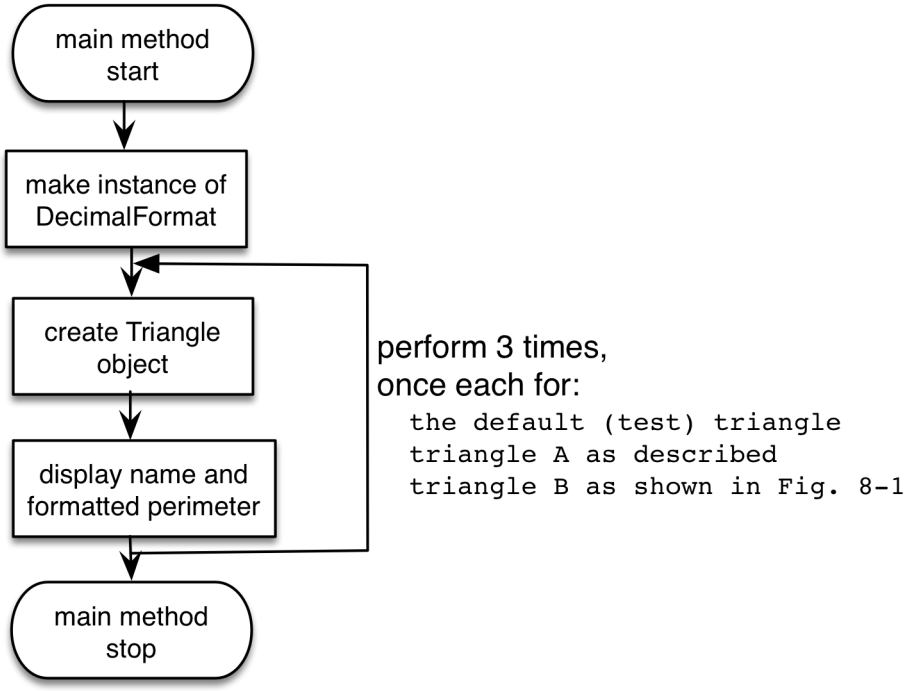
Plans for Lab Work

The UML class diagram for the class `Triangle` as described in the Lab Work



The UML class diagram for the application class (which isn't much use)

A flow diagram showing the sequence of events required in the `main` method.



Lab Work

Part 1 Specification:

Write an application that calculates the perimeter of 3 triangles, given the coordinates for the three corners of each. Use the UML specification on page 43. Note there is no `toString` method. The triangles are drawn for you in Figure 8-1. The TEST case is created by the default constructor. Triangles A and B are created by sending their coordinates and names to second Triangle constructor.

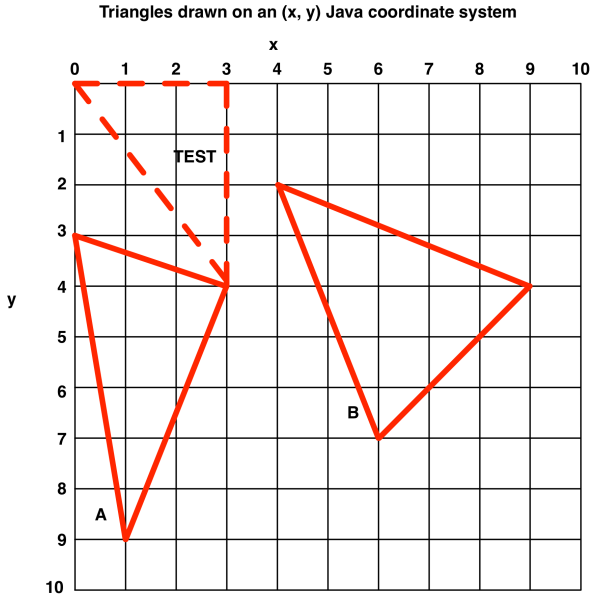


Figure 8-1

Compute the length of a triangle's side by calculating the distance between two points (corners) using the following formula:

$$distance = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$$

The perimeter of a triangle is the sum of the 3 distance calculations, 1 for each side.

The default test case (0,0) (3,0) (3,4) represents a right-angled triangle where the legs (sides next to the right angle) are 3 and 4 units long respectively, and the hypotenuse (side opposite the right angle) will be 5 units long according to the formula above. Therefore the perimeter will be 3 + 4 + 5 = 12 units.

The UML class diagram for the **support class** (Triangle) is shown on the page 43. Each instance of the support class represents one triangle. The following paragraphs describe what the UML diagram sums up more succinctly.

- The Triangle class has data fields to store the coordinate and name information for the triangle.
- The default constructor takes no parameters, and sets the data fields literally using the test data.
- The second constructor takes the data in as parameters, and uses these parameter values to set the data fields
- The Triangle class has a method to calculate and return the length of **one** side using the `distance` formula above. This method should take 4 `int` values as parameters, representing the x , y values for each end of the side. The UML diagram insists they should be called x1, y1, x2 and y2 to match the names used in the distance formula.
- The Triangle class has a method to add the lengths of its 3 sides together and return the perimeter. This method will call the `calcSide` method 3 times, each time sending it a different set of 4 integers representing a different side of the triangle. The values returned by these 3 method calls will be added, and the resulting sum returned.
- The Triangle class should have an accessor method which returns the name of the triangle.

The **application class** (TriangleApp) should create the test Triangle object using the default constructor, and 2 further Triangle objects by sending co-ordinate data to the constructor e.g.

```
Triangle a = new Triangle(0,3,3,4,1,9,"A");
```

The application class should display name and the perimeter of each triangle, formatted to 2 decimal places.

```
e.g. Triangle A perimeter is 14.63 units
```

BEFORE YOU START CODING Look at the UML class diagrams and flowchart provided. Read and re-read the specifications above to make sure you understand the task. What data fields are required? What type of data will they store? Which data needs to travel from one class to the other via the constructor? What methods are required? Will they be void or return? When/how often will they be called? What parameters will they need? Does it matter which direction you go around the triangle? Does it matter which point you start with?

Extension:

Make the Triangle class calculate the length of the hypotenuse if it is given the length of the 2 legs of a right angle triangle rather than the 3 coordinate pairs.

What changes would you need to make if the corners weren't exactly on the grid intersections i.e. the coordinates aren't necessarily whole numbers?

Part 2

Write a program which displays a random integer within a specified range. The application class should read the high and low integer values required from the user, create an instance of the support class, and display the result of a call to your random range method. The support class should contain no data fields and just one method, which takes two `int` parameters (representing the high and low limits of the random range) and returns a random integer between (and inclusive of) the two parameters.

You will find listed below (and also in a file called `code.txt` in the **LabFiles> Lab08** folder) 2 comment blocks, 2 import statements, 2 class definitions and 3 methods. **This is all the code you will need.** Your job is to arrange this code into an application class and a support class as described above. The block of code within each box stays together.

```
/** RandomApp.java    Lab 8, Part 2, COMP160  2019
 * Displays a random integer between high and low limits
 * High and low values are typed in by the user.
 */

/** RandomRange.java   Lab 8, Part 2, COMP160  2019
 * Contains a single method which returns random integer between high and low parameters.
 */

import java.util.Scanner;
import java.util.Random;
public class RandomApp{
}
public class RandomRange{
}
```

```
public static void main(String[]args){
  int lo = readInt("Enter lowest value");
  int hi = readInt("Enter highest value");
  RandomRange r = new RandomRange();
  System.out.println("Random integer between " + lo + " and " + hi + ":" +
                                                    r.randomRange(lo, hi));
}
```

```
/** Returns random integer between high and low parameters.
  @param low lowest value of range required
  @param high highest value of range required
  @return a random integer between low and high  values */
public int randomRange(int low, int high){
  Random generator = new Random();
  return generator.nextInt(high – low + 1) + low;
}
```

```
/** Returns an integer entered by the user
  @param message a prompt to the user
  @return an integer typed by the user*/
public static int readInt(String message){
  Scanner sc = new Scanner(System.in);
  System.out.println(message);
  return sc.nextInt();
}
```

Lab 8 Completed

The code I present for my Lab 8 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete	☐ comments	☐ Triangle class as per UML diagram
☐ output formatted to 2 DP	☐ random	☐ submitted

Date **Demonstrator's Initials**

Laboratory 9 Selection 1

Notes

Readings: L,D&C Chapter 4, Sections 4.1, 4.2, 4.3 (pages 112 – 130).
Selection is a fundamental part of all programming languages - we need to be able to select what to do next under particular conditions. This lab gives you practice formulating the conditions, which are boolean expressions and introduces the `if else` structure. You will get more practice writing constructors, formatting output and working with objects.

The logic for what appears to be a simple task may get quite complicated. Careful thought and planning are required, but that is not the end of it. After all that work, even if your code is producing answers, it will still need thorough, rigorous testing.

Preparation

1. Calculate the result of each expression if `a` is `5`, `b` is `9`, `c` is `14`, and `flag` is `true`.

a. `a == ( b + a – b)`

b. `( c == ( a + b ) ) || ! flag`

c. `( a != 7 ) && ( c >= 6) || flag`

d. `! ( b <= 12 ) && ( a % 2 == 0 )`

e. `! ( ( a > 5 ) || ( c < ( a + b ) ) )`

2. Given:
```
char ch = 'f', digit = '8';
String aString = "comp";
```
What does each of the following evaluate to?

a. `'a' <= ch && ch <= 'z'`

b. `digit > '0' || digit < '8'`

c. `aString.equals("comp")`

d. `"comp".equals(aString)`

e. `!aString.equals("comp")`

3. Write statements which will swap the values stored in `low` and `high` if `low` is larger than `high`.

```
Scanner scan = new Scanner(System.in);
int low = scan.nextInt();
int high = scan.nextInt();
int tempStore = 0;
```

4. Write a boolean expression for each of the following:

a. `year` is divisible by 4

b. `waterLevel` is greater than 2.5 and less than 3.9 inclusive

c. `ch` is an uppercase letter

5. Write a `boolean` expression to assign a value of `true` to `child` if `age` is in the range `0` to `17`, inclusive; otherwise assign a value of `false`.

6. Write a `boolean` expression to assign a value of `true` to `pass` if `score` is in the range `50` to `100`, inclusive; otherwise assign a value of `false`.

7. What output is produced by the following code fragment?

```
int limit = 100, num1 = 15, num2 = 40;
if (limit <= 200){
   if (num1 == num2)
      System.out.println("lemon");
   System.out.println("lime");
}
System.out.println("grape");
```

8. The two code fragments below perform the same task. Which is easier to read?

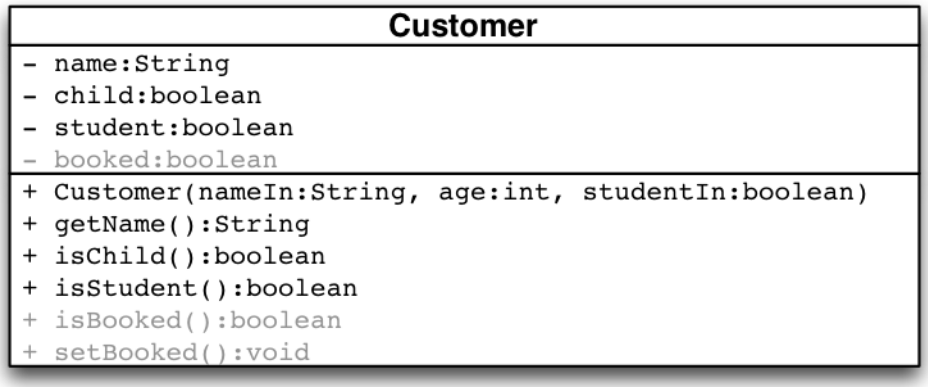
```
if( score > 50){
    if ( score < 100){
      printCertificate();
    }
 }
```

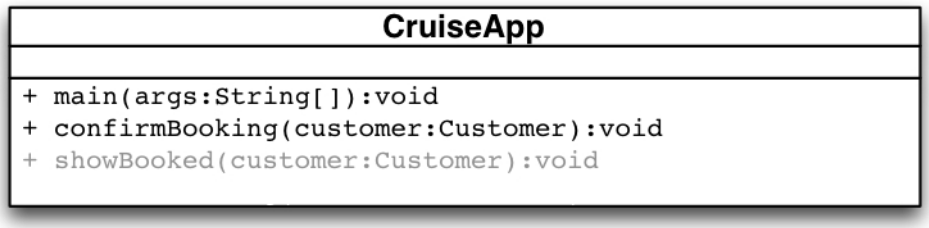
```
if (score > 50 && score < 100){
     printCertificate();
 }
```

Plans for Lab Work

Members in black are described in the initial setup of steps 1 and 2.

Members in gray are described in the later steps, 5 and 6.





Lab Work

If you already know what an array is, and how to use a loop, or are comfortable enough to look ahead to these concepts, please feel free to use them in your lab work.

The process of writing classes as described below uses incremental development, where a piece of program is developed and tested, then another piece of code or more functionality is added. After each step the program should compile, and (as soon as the main method is included) run. This method of breaking tasks down into successively smaller tasks is called a top–down approach.

A class (called `Customer`) will represent a potential customer for a meal and discounted harbour cruise package. Various discounts will be offered for children and students.

The application class will create instances of the `Customer` class, and calculate the costings according to the `child`/`student` status of each `Customer` object.

Since we have not covered reading from files yet, the data for creating each instance of `Customer` is provided for you as constructor calls in `CruiseApp`, the application class starter provided in the **coursefiles160** folder and printed below. The customer's name, age and a boolean value for having presented a student ID card are the information given.

```
public static void main(String[]args){
        // each Customer created with name, age, showed student ID card
        Customer customer1 = new Customer("Aaron Stott",17, true);
        Customer customer2 = new Customer("Betty Adams",17, false);
        Customer customer3 = new Customer("Corin Child",16, true);
        Customer customer4 = new Customer("Doris Stewart",25, true);
        Customer customer5 = new Customer("Edmond Cheyne",12, false);
        Customer customer6 = new Customer("Fiona Chaney",7, false);
        Customer customer7 = new Customer("Ged Still-Child",16, true);
        Customer customer8 = new Customer("Harry Adamson",20, false);
        confirmBooking(customer1);   //and so on
}
```

Fig 9.1 The main method of CruiseApp.java, provided in coursefiles160

1. From this starting point, and with the help of the UML class diagrams, plan and write a `Customer` class which stores the name and two `boolean` data fields, one for whether the customer is a student, the other for whether the customer is a child. A child is defined as between 5 and 16 years (inclusive).

The class should have accessors for these three data fields. Rather than prefix the boolean accessor's name with `get`, use `is` e.g. `isChild()`. The advantage of this will be easier-to-read `if` statements later on.

2. The main method will create `Customer` objects when run. Write another method in the application class called `confirmBooking`. This method will eventually display prices and confirm the booking. It will need to take a `Customer` object as a parameter. Initially, to test that your structure is working so far, just get this method to print out the data returned by the accessors. You will need to call this method 8 times from the main method, once for each `Customer` object (unless you already know about loops and arrays, in which case you could save yourself some typing).

```
Aaron Stott false  true  (the spaces won't be there unless you have explicitly requested them)
Betty Adams false  false
Corin Child true true
Doris Stewart false  true
Edmond Cheyne true false
Fiona Chaney true  false
Ged Still-Child true true
Harry Adamson false  false
```

3. Now that the structure is working, you can concentrate on what happens inside the `confirmBooking` method.

This method should

- specify that a standard ticket price is 56.0 and a standard meal price is 30.0.
- calculate discounts .
  - ► Students and children receive half price tickets, but anyone else gets a 20% discount.
  - ► Children receive half-price meals. Everyone else gets a 10% discount.
- display the prices and the total for each customer.

```
Aaron Stott    Ticket price:28.0    Meal price:27.0   Total price:55.0
Betty Adams    Ticket price:44.8    Meal price:27.0   Total price:71.8
Corin Child    Ticket price:28.0    Meal price:15.0   Total price:43.0
Doris Stewart   Ticket price:28.0   Meal price:27.0   Total price:55.0
Edmond Cheyne   Ticket price:28.0   Meal price:15.0   Total price:43.0
Fiona Chaney    Ticket price:28.0   Meal price:15.0   Total price:43.0
Ged Still-Child Ticket price:28.0   Meal price:15.0   Total price:43.0
Harry Adamson   Ticket price:44.8   Meal price:27.0   Total price:71.8
```

4. Format the price information as you would expect it to be displayed.

```
Aaron Stott    Ticket price:$28.00   Meal price:$27.00 Total price:$55.00
```

5. So far, so good. The next component of the task is to present the information tidily, one customer at a time, and ask for confirmation of the booking. The user should confirm by pressing **`y`** or **`Y`**.

The booking information will need to be stored. This will require another `boolean` data field in the `Customer` class. Write a mutator and an accessor for it.

Back in the application class, finish your `confirmBooking` method. If the choice is **y** or **Y**, set the customer's `booked` field to `true` and display `"Booked"`. (Refer back to Preparation Question 2 c, d and e for how to compare the content of 2 strings for equality. The string containing the choice and the string `"y"` will not be references to the same object so == will not be useful.)

```
Aaron Stott
Ticket price: $28.00
Meal price:   $27.00
Total price:  $55.00
Confirm booking for Aaron Stott(Y/N)
Y
Booked
```

6. Lastly, the business would like a listing of those who have booked.

In the **application** class, write another method which takes an instance of `Customer` as a parameter. All this method should do is print out the name of the customer if (and only if) they are booked. You will need to call this method 8 times (once again, we are looking forward to having loops and arrays to make our job easier).

The method's print out might look something like this:

```
Aaron Stott is booked
Corin Child is booked
Edmond Cheyne is booked
Ged Still-Child is booked
```

Lab 9 Completed

The code I present for my Lab 9 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 10 Selection 2

Readings: L,D&C Chapter 4, Sections 4.2 and 4.4 (pages 116 – 127, 130 – 134).

Preparation

1. What will be displayed after each of the following conditional statements have been performed?

(You will need to refer to L,D&C Section 4.2 The Conditional Operator page 124)

```
int a = 7;
int b = 8;
int c = 2;
System.out.println( (a < b) ? a : b);          _____________

System.out.println( (a == b) ? a : b);         _____________

System.out.println( (a < b) ? a + c : a – c);  _____________
```

2. For what value(s) of `a` will the condition `(a > 0 || a < 0)` be `false`?

3. For what value(s) of `a` will the condition `(a < 100 || a > 0)` be `true`?

4. Pick 5 values for `a` which would test the condition in Q3. Then check your answer.

1. 2. 3. 4. 5.

5. Explain the difference between

```
a = 0;
if (b > 0) a++;
else if (c > 0) a++;
```

and

```
a = 0;
if (b > 0) a++;
if (c > 0) a++;
```

6. What will the output of this code be if `a` has the value 25?

```
if (a > 10){
     System.out.println("a is greater than 10");
} else if (a > 20){
     System.out.println("a is greater than 20");
}
```

7. In the two code fragments below, tick the statements which will be performed if `a` has a value of `12`

1.
```
if(a < 10)
  b = 5;
  c = 6;
  System.out.println(a);
```

2.
```
if(a < 10){
   b = 5;
   c = 6;
}
System.out.println(a);
```

Plans for Lab Work

Lab Work

Programming Projects PP 4.1 (L,D&C page 163). Design and implement an application that processes a series of integer values representing years. The purpose of the program is to determine if each year is a leap year (and therefore has 29 days in February) in the Gregorian calendar. A year is a leap year if it is divisible by 4, unless it is also divisible by 100 but not by 400.

For example, the year 2018 is not a leap year, but 2020 is. The year 1900 is not a leap year because it is divisible by 100 but not by 400. The year 2000 is a leap year because it is divisible by 100 and also divisible by 400. Produce an error message for any input value less than 1582 (the year the Gregorian calendar was adopted). This information is summarised in Table 10-1.

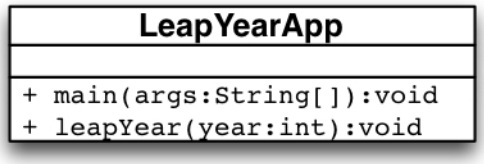
The application should display the result to the console window, as below

```
2018: is not a leap year
2020: is a leap year
1900: is not a leap year
2000: is a leap year
1565: predates the Gregorian calendar
```

	/4	/100	/400	Leap
2018	no	no	no	**no**
2020	yes	no	no	**yes**
1900	yes	yes	no	**no**
2000	yes	yes	yes	**yes**

Table 10-1

Just one class is sufficient for this task. The code, however, should not be without structure.



The statements which "process" the year and display the output should be in a method which takes each individual year as a parameter.

The `main` method should call this method at least 5 times, once for each of the 4 "test case" years listed in Table 10-1 above, and once more for a year before 1582. e.g.

```
leapYear(2018);
leapYear(2020);
leapYear(1900);
leapYear(2000);
leapYear(1565);
```

Lab 10 Completed

The code I present for my Lab 10 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments

☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 11 Strings

Notes

Readings: L,D&C Chapter 3, Section 3.2 (pages 80 – 83).

While `int` and `double` are primitive data types, `String` is a reference data type. `String` is a class in the Java libraries (API). When you create a `String` you are creating an instance of this class. The String class contains many methods which may be called on any `String` object. This lab will explore some of them.

Background

Think of a `String` as a series of characters stored in a sequence. Each character is in its own position and has an index number. The numbers start at 0. (See the diagram below.)

A String has a name. `String animal = "tree frog";`

`String colour = "red";`

Knowing the length of a `String` may be very useful in some circumstances. The `String` class has a method `length` which returns the length of the string. It can be called like this:

```
int stringLength = colour.length(); //assigns the length of a String called colour to an int variable
System.out.println(colour.length()); //displays (doesn't store) the length of the String called colour
```

The length of `colour` is 3. The characters are at index positions 0, 1 and 2.

- The length returned by the `length()` method is the count of characters as humans count, starting at 1. It is an integer.
- Each character has a position or index in the string. The index starts at `0` and finishes at `length() - 1`.
- A space is a character like any other.

The String declaration: `String songLine = "While my guitar gently weeps";` is the basis for the diagram below.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Character	W	h	i	l	e		m	y		g	u	i	t	a	r		g	e	n	t	l	y		w	e	e	p	s

Trying to access an index out of the valid range will produce a `"StringIndexOutOfBoundsException"` error when the code is run. In DrJava this appears in the Interactions pane in red. (This is a run-time error.)

For the declaration above, `songLine.length()` will return the value _____

For the declaration above, the last accessible index position of `songLine` is _____

The `String` class has many methods which can be called. To access the methods, we use dot notation to identify which particular `String` the method is being called on e.g.

```
animal.length()  colour.indexOf('e')  songLine.charAt(3)  animal.substring(2,4)
```

The text book describes some of these String methods in Section 3.2. A complete list can be found in the APIs/libraries.

> The `String` class is in the `java.lang` package which is imported automatically to all Java programs. When a statement like `String word = "duck"` is run, an object of type `String` is created.
>
> If a variable is an instance of a Java class its data type is a **reference** type. Reference types refer to an object. A reference variable doesn't hold the object itself. It holds the memory address of the object (where it can be found in memory).
>
> The primitive data types are of known size for storing in computer memory. A reference data type does not have a set size. This is clearly demonstrated by the String class. A `String` may refer to a single character or thousands of characters.

Preparation

1. What value is stored in the variable after each of the following statements have been executed? Assume the declaration

```
String drink = "Ginger ale";
```

The API shows good examples of substring being used. Remember that the length of a string is the number of characters it contains (e.g. 10 for `drink` shown above) but the index positions start at 0.

index	0	1	2	3	4	5	6	7	8	9
character	G	i	n	g	e	r		a	l	e

a. `char c = drink.charAt(0)`

b. `char c = drink.charAt(drink.length() - 1)`

c. `String s = drink.substring(1)`

d. `String s = drink.substring(3, drink.length() - 2)`

The String method `indexOf` isn't listed in Section 3.2 (L,D&C), but it is in the Java library documentation. For your convenience, here are the descriptions:

> **`public int indexOf(int ch)`**
> Returns the index within this string of the first occurrence of the specified character.
> **`public int indexOf(int ch, int fromIndex)`**
> Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
> i.e. starts searching at `fromIndex`

Which of these methods is called depends on the parameters used in the method call. A single `char` or `int` e.g. `indexOf('k')` will call the first version. (This is called **method overloading** and is discussed later.)
A `char` or `int` followed by an `int` e.g. `s.indexOf(97,3)` will activate the second version.

These methods can be sent either a character e.g. **`'e'`** or a Unicode value e.g. **`101`**. See Appendix C for a listing of common Western unicode characters and their numeric values.

```
String s = "cabbage";
System.out.println(s.indexOf(97));   //these 2 lines of code
System.out.println(s.indexOf('a'));  //perform the same function
```

e. `int first_e = drink.indexOf('e')`

f. `int second_e =  drink.indexOf('e', 5)`

index	0	1	2	3	4	5	6	7	8	9
character	g	r	e	e	n		t	e	a	.

g. `The` variable `second_e` is intended to store the position of the second `'e'found in the string`.
If the string stored in `drink` is changed to `"green tea."`, would the statement
`second_e =  drink.indexOf('e', 5)` still find the second `'e'`? _____

The **`5`** in part **f.** is not flexible – it is `"hardwired"` and will not adapt if the String is changed.
Adapt the expression used in **f.** so that it finds the position of the second `'e'` in the String `drink` in terms of the position of the first 'e'.

```
int second_e =  drink.indexOf('e',              )
```

h. `drink.toUpperCase()`

i. `drink.toLowerCase()`

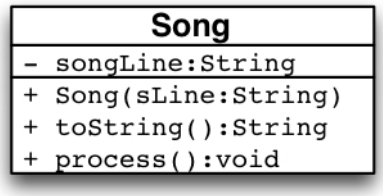
2. Look up the `String` class in the library documentation - it is in the `java.lang` library. Refer back to Lab 3, Part 2 (page 21 for instructions on accessing the library (Java API) documentation. Scroll down until you find its method `length`.

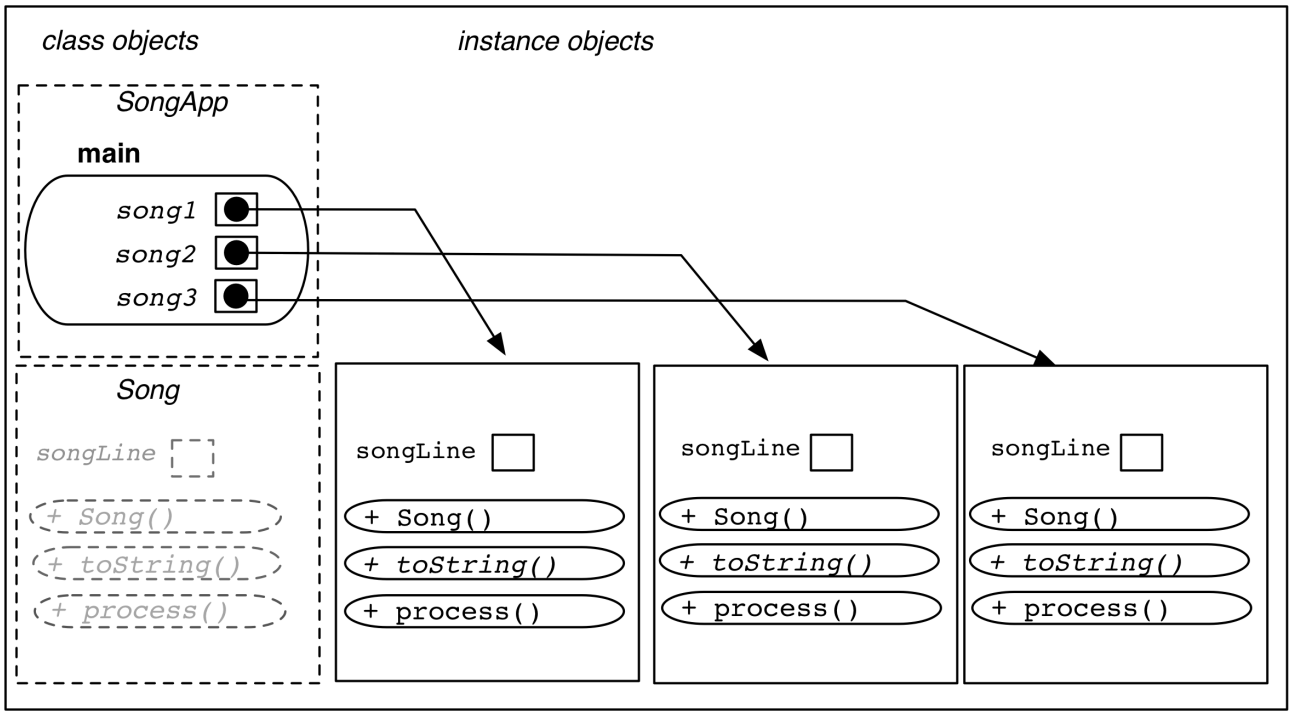
a. What is the short definition given?

b. What is the return type of `length`?

3. Given the statement `String title = "Modern Times";` write a statement which will declare a variable called `titleLength` and assign to it the length of the string called `title`.

Planning for Lab Work





Lab Work

Note: it is not a good idea to call any class you write `String` because this will prevent your code from accessing the `String` class already written in the APIs.

1. Make a new class called `Song` as described by the UML diagram on the previous page. The **constructor** sets the value of the data field and the `toString` method returns the value stored in the data field.

2. Still in the `Song` class, write a `void` method called `process()`. The method can remain empty for now.

3. Make a new application class called `SongApp` (using `App` in the application class's name helps people to locate the `main` method in programs involving more than one class). In the `main` method:

- create an instance of `Song`, initialising the data field to "`While my guitar gently weeps`".
- display the value stored in the data field of your instance object. You cannot access this data directly from another class because it is private, but you can access it through the public `toString` method which returns its value.
- write one more statement which calls the `process` method.

If you compile and run now you should just see the String stored in the data field displayed.

4. Still in the `main` method, make another instance of `Song`, initialising its data field to "`Let it be`". As before, display the value stored in this instance's data field and call its `process` method.

The structure of your program is now in place. The application class creates two instance objects, sends them data, and calls their methods either to perform tasks or retrieve information. You can think of the application class as a manager class, whose job is to create instances of other classes and issue orders which organise the data in and out of them.

All the remaining code is written in the `process` method of the support class.

5. Print out the length of `songLine`. Label the output rather than just showing a number e.g. `Length is: 9`.

6. Print out the last character in `songLine` using `charAt`.

7. Your next task is to print the first two words of the `songLine` data field on one line and all subsequent words on the next line.

(Make sure you have completed and understood Preparation Exercise 1 to gather the knowledge you need for this task. You are welcome to use local variables for storing information. Use descriptive names to aid readability.)

8. Print out the first letter of the third word of `songLine`.

9. Print out `songLine` in uppercase. Use `toUpperCase`.

10. Print out `songLine` with the spaces replaced by the letter '`x`'.

11. Use `indexOf` to find the position of the first occurrence of the letter `'b'` in `songLine`. Display the result in a `println` statement. What is the integer returned if the target character is not found in the string?

_____

12. Has the value stored in `songLine` changed as you have worked through the exercise above? Print out `songLine` to check your answer.

_____

13. Make one more instance of the `Song` class, this time sending `"Penny Lane"` to the constructor. Run your code again. You will see a run time error. Use an `if` statement wherever necessary to avoid attempting to access index positions which don't exist.

Lab 11 Completed

The code I present for my Lab 11 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments ☐ embedded questions answered
☐ descriptive names, output labelled ☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 12 Repetition 1

Notes

Readings: L,D&C Chapter 4, Sections 4.5, 4.6, 4.7 (pages 134 – 151).
Much of the power of computing lies in the ability to repeat tasks a certain number of times, or until a particular state has been reached, or until the end of the job. Loops are the key to repetition, and there are a number of different loops which may be used. In this lab we will look at the `while` and the `do-while` loops.

Preparation

1. Exercise EX 4.7 (L,D&C page 161). What output is produced by the following code fragment?

```
int num = 0, max = 20;
while (num < max){
   System.out.println(num);
   num += 4;
}
```

2. Exercise EX 4.8 (L,D&C page 161). What output is produced by the following code fragment?

```
int num = 1, max = 20;
while (num < max){
   if (num % 2 == 0)
      System.out.println(num);
   num ++;
}
```

3. Write a `while` loop which counts the number of spaces in a String called `mySentence`. Declare any variables required.

4. What is the minimum number of times a `while` loop is executed?

5. What is the minimum number of times a `do-while` loop is executed?

6. Adapted from Exercise EX 4.19 (L,D&C page 163).

a. Finish the code below so it reads exactly 10 integer values from the user and prints the highest value entered.

```
import java.util.Scanner;
public class HighApp{
    public static void main(String[]args){
        Scanner scan = new Scanner(System.in);
        int count = 1;
        System.out.println("Enter number " + count);
        int highest = scan.nextInt(); //first, assume highest
        count++;
        while(_____________){ // loop continues while the condition remains true
            System.out.println("Enter number " + count);
            int input = ____________________
            if (input > highest){
                ________________________
            }
            count++;
        }
        System.out.println("The highest number entered was " + highest);
    }
}
```

b. Why is the initial value of `highest` set to the first input value rather than 0?

7. The code in Question 6 is a counter controlled loop. To turn this loop in to a state controlled loop, finishing and reporting its message when `highest` reaches 80 or more, how would you write the while condition?

```
while(_________________________){ // loop continues while the condition remains true
```

Plans for Lab Work

Lab Work

These programs can both be written just in the main method of an application class.

Part 1

Programming Projects PP 4.3 (L,D&C page 163).

Design and implement an application that reads an integer value and prints the sum of all even integers between 2 and the input value, inclusive. Print an error message if the input value is less than two. Prompt the user when requesting input, and label the output clearly.

A typical output might be:

```
Enter an integer greater than 1
  7
Sum of even numbers between 2 and 7 inclusive is: 12
```

or

```
Enter an integer greater than 1
 -3
Input value must not be less than 2
```

Part 2

Adapted from Programming Projects PP 4.12 (L,D&C page 165).

Design and implement an application that reads a string from the user, then determines and prints the number of vowels and consonants which appear in the string.

Hint: Use a `switch` statement inside a loop. If you omit the word `"break"` between cases, you can list several cases and do the same thing for all of them e.g. for a variable called `grade`, of type `char`, which has already been declared and assigned a value:

```
switch(grade){
   case 'A':
   case 'B':
   case 'C':
      System.out.println("Pass");
      break;
   case 'D':
   case 'E':
      System.out.println("Fail");
      break;
   default:   //if the character stored in grade is not listed as a case, this will apply
      System.out.println("Uncoded input");
}
```

A typical program output for this exercise might be:

```
Enter a sentence
 My dog has fleas!
Sentence is : My dog has fleas!
VowelCount : 4
ConsonantCount : 9
```

Lab 12 Completed

The code I present for my Lab 12 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments ☐ Part 1 sum
☐ Part 2 vowels ☐ submitted

Date **Demonstrator's Initials**

Laboratory 13 Repetition 2

Notes

Readings: L,D&C Chapter 4, Section 4.8 (pages 151 – 157. Reread L,D&C Chapter 4 Section 4.3 (pages 127 – 130).

The `for` loop is particularly useful for the systematic processing of a series of related data. In this lab you will use a `for` loop to access each character in a `String`.

Preparation

1. Write a `for` loop that displays the following set of numbers:

0, 10, 20, 30, 40, 50 . . . 1000

2. Write a `for` loop to print the odd numbers from 99 down to 1 (inclusive).

3. Finish the `for` loop which prints out all the ASCII characters from `'A'` to `'z'` on one line, with a space between each.

```
for(char c = 'A';                          ){

    System.out.print(              );

}
```

4. What output will the nested for loop below produce?

```
for(int i = 0; i < 3;i++){
    for(int j = 0; j < 3;j++){
        System.out.print("*");
    }

    System.out.println();
 }
```

5. What output will the nested for loop below produce?

```
for(int i = 0; i < 3; i++){
    for(int j = 0; j <= i; j++){
        System.out.print("*");
    }

    System.out.println();
}
```

6. Write a loop which will print every character of a String followed by a # character e.g. the String "Ten umbrellas" should be displayed thus: T#e#n# #u#m#b#r#e#l#l#a#s#

```
String s = "Ten umbrellas";
```

7. Write a nested loop which produces the output

```
AAAA
BBBB
CCCC
DDDD
```

8. Adapt the loop from the previous question so it produces the output

```
ABCD
ABCD
ABCD
ABCD
```

Planning for Lab Work

Design overview – to produce a sorted string (without using any data structures not yet covered in the course)

make the phrase lower case

make a new empty string which will be built up into the sorted string

find every `'a'` in the lower case phrase and append it to the sorted string

find every `'b'` in the lower case phrase and append it to the sorted string

find every `'c'` in the lower case phrase and append it to the sorted string

etc.

Algorithm:

get first phrase from user
make the phrase lower case
create a new empty string (ready to store each letter in alphabetical order)
for each letter of the alphabet (a to z)
for each index position in the phrase string
if the char at that position matches the letter, append it to the new string
(first it will find all the a's, then the b's etc. Punctuation, digits, spaces etc. will be ignored.)

repeat the process for the second phrase (the process could be in a method which is reused for each phrase)

compare the 2 sorted strings – if they are equal, they are anagrams. (remember Lab 9, Prep 2 c, 2 d, 2 e)

Lab Work

An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one phrase is an anagram of another phrase. The program should ignore white space and punctuation. The phrases should be typed in at the keyboard.

This program can just be written in the main method of an application class.

Typical expected output:

Enter first phrase

replays

Enter second phrase

parsley

aelprsy are the letters of replays in order

aelprsy are the letters of parsley in order

replays is an anagram of parsley

Some more word pairs and phrases to test with:

Resistance Ancestries	Gainly Laying	Admirer Married	Orchestra Carthorse
Creative Reactive	Deductions Discounted	Listen Silent	
Paternal Parental	Angered Enraged		

A highwayman	Away! Hang him!
Internal Revenue Service	I never return even a slice.
Shakespeare	Seek a phrase.
Received payment	Paid me every cent.

Lab 13 Completed

The code I present for my Lab 13 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 14 Graphical Objects

Notes

Readings: L,D&C Appendix F (pages 966 – 976) and Chapter 3, Section 3.8 (pages 100 – 102).
It is vital for this lab that you read and understand the text book pages covering graphical objects and the `paintComponent` method.
You have already created graphical objects - in Lab 6 you made objects of the type `MyPanel`. In this lab, you will work with three classes: an application class and two support classes. One of the support classes will create and work with several instances of the other support class.

Preparation

1. Given the two classes below, what will be printed out when `TestApp` is run?

```
public class TestApp{
  public static void main(String[]args){
    Test t = new Test();
    System.out.println(t.toString());
  }
}
```

```
public class Test{
  private int x;
  private int y;

  public Test(){
    int x;
    x = 3;
    this.x = 4;
  }
  public String toString(){
    return("The value of x is " + x );
  }
}
```

2. A series of statements in some class looks like this:

```
SecondClass s1 = new SecondClass();
SecondClass s2 = new SecondClass();
System.out.println(s1.toString());   //if no toString method has been written, these will
System.out.println(s2.toString());   //print out the object's memory address (reference)
s1 = s2;
System.out.println(s1.toString());
System.out.println(s2.toString());
```

If the first two `println` statements produced the output

```
SecondClass@b890dc
SecondClass@123c5f
```

what will the output of the other two `println` statements be?

3. The `Graphics` method `drawString` requires three inputs - the string to be 'drawn' and 2 integers representing the coordinates. The code

```
int i = 3, x = 100, y = 100;
g.drawString(i, x, y);
```

will fail. Why?

4. The `Integer` wrapper class in `java.lang` has a `toString` method which converts the integer to a string. Rewrite the `drawString` statement from Ex. 3 above so that the code will display a string representation of `i`.

```
g.drawString(
```

Plans for Lab Work

Diner
- x:int - y:int - name:String - seatNum:int - colour:Color - DIAMETER:final int = 50
+ Diner(x:int,y:int,name:String,seatNum:int) + draw(g:Graphics):void

TablePanel extends JPanel
- diner1:Diner - diner2:Diner - diner3:Diner - diner4:Diner - diner5:Diner - diner6:Diner
+ TablePanel() + paintComponent(g:Graphics):void

Lab Work

Programming Projects PP F.21 (L,D&C page 984). Write a program that displays a graphical seating chart for a dinner party. Create a class called `Diner` (as in one who dines) that stores the person's name, and location at the table. A diner is graphically represented as a circle, with the person's name printed in the circle and place number to the top. The circles are coloured so they alternate around the table – odd seat numbers are white, even seat numbers are gray.

Assume there will be 6 guests around the table. The `Splat` program listing (Appendix F listing F.3, F.4, F.5 pp 973 – 976) will be a useful example to follow.

Your program should produce a window something like the picture in Figure 14.1. Don't waste too much time on text positioning - that is not the purpose of this exercise.

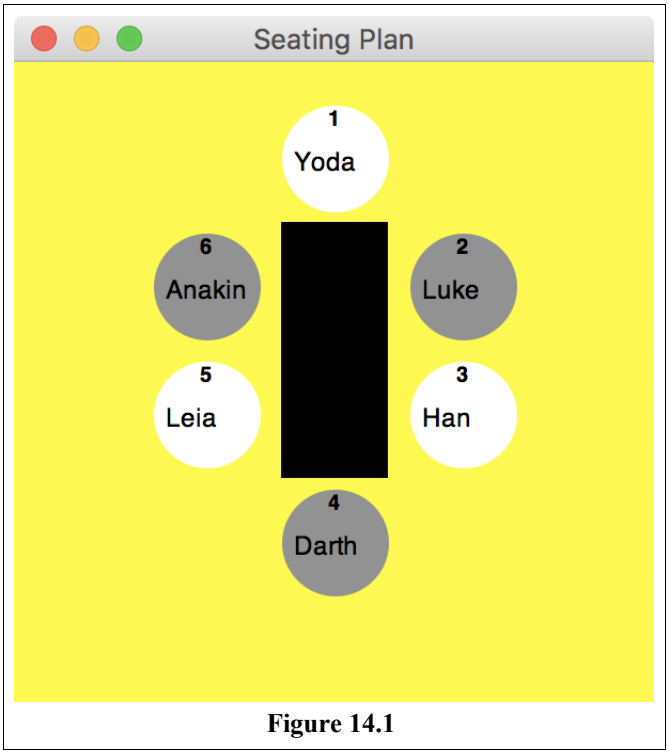


Figure 14.1

In the `Splat` class the `main` method creates the `JFrame` and adds not just a `JPanel` but a user-created extension of `JPanel` called `SplatPanel`.

All the behaviours of the `JPanel` class (such as `add, paintComponent, setBackground` . . .) are available to `SplatPanel` because of the keyword `extends` in its class header.

Each circle is an instance of the Circle class, stored in `Circle` data fields, created using `Circle` constructors and drawn in the `paintComponent` method using calls to each circle's `draw` method.

Specifications for a Solution for Diner

The first thing to do **before you go any further** is work out the coordinates for your drawing elements - each of the 'diner' circles and the table. You will need to have set a size for your drawing window/panel (we used 300 by 300) and a diameter for your circles (we used 50). Write the coordinates for each element on Figure 14.1 (previous page).

All the graphics drawing in the program will be handled by the `paintComponent` of a `JPanel`, in this case the `TablePanel` class (which extends `JPanel`) described below. The `paintComponent` method of `TablePanel` will call the `draw` method on each `Diner` object, passing it a reference to its `Graphics` object so that `Diner` can draw itself in the `JPanel` graphics context.

The Diner class ***(compare with the Circle class L,D&C Listing F.5 page 976)***

The `Diner` class represents a single diner by means of a filled circle. The circle has an x,y location, a name, a colour representing gender and a number representing the seating position. All this information is sent to it when it is created, then stored in its data fields. Because the diameter of the circle is the same for each diner, it can be set literally in the `Diner` class. It should also be a constant, as it doesn't change. The `Diner` class contains a method to draw itself using the data in its data fields.

1. The `Diner` class will need data fields representing:

- an x location
- a y location
- the name of the diner
- the colour of the circle (the class will need to import `java.awt.*` in order to use `Color`)
- the number of the seating position
- the diameter of the circle, set to a standard size

2. The `Diner` class will need a constructor which

- sets the data fields representing x and y location, name and seating position to the value of its parameters.
- sets the colour according to whether the seating position is odd (white) or even (gray).

3. The `Diner` class will need a method (call it `draw`) to draw itself which

- takes a graphics object as a parameter
- sets the drawing colour to the colour held in the data field
- draws the circle at the x,y location
- sets the drawing colour to the colour required for text
- sets the font face and size for the name (see the box on the next page for tips on setting a font)

> **Note**: The `drawString` text is drawn **above** the `x` value given, not below as is the case for `drawOval` and `drawRect`. From the API:`drawString(String str, int x,int y)` Draws the text given by the specified string, using this graphics context's current font and color. The baseline of the leftmost character is at position (*x*, *y*) .

– prints the name at a location relative to the circle. **(**Long names will spill out of the circle.)

- sets the font face and size for the seating position. **Note:** the font size for the place number is smaller than that for the names.

- prints the seating position at a location relative to the circle

Tips for Setting a Font

The code

```
xxx.setFont(new Font("Courier", Font.PLAIN, 8));
```

sets the font for a `Graphics` object called `xxx` to `Courier, plain, 8 point`,

or you can set the font by creating an instance object of type `Font`:

```
Font boldH14 = new Font ("Helvetica", Font.BOLD, 14);
```

and using it later on a particular `Graphics` object e.g.

```
page.setFont(boldH14);
```

4. The `Diner` class will need to import `java.awt.*` in order to use `Color` and `Graphics`.

This class is now complete, so this could be a good opportunity to compile it to check for errors.

The TablePanel class *(compare with the SplatPanel class L,D&C Appendix F Listing F.4 page 975)*

The `TablePanel` class needs to be an extension of the `JPanel` class (it wants to inherit all of the methods of `JPanel`, as well define some extra ones that you will write yourself). It creates the 6 `Diner` objects and calls their `draw` method, as well as drawing the `JPanel` and the rectangle representing the table.

1. The `TablePanel` class will need to extend `JPanel`.

2. The `TablePanel` class will need to declare data fields to hold references to 6 `Diner` objects.

3. The `TablePanel` class will need a constructor which

- creates 6 `Diner` objects, sending each appropriate values
- sets the size and background colour of the panel

4. The `TablePanel` class will need a `paintComponent` method which

- takes a `Graphics` object as an input parameter
- draws a panel of a size and background colour by calling the `paintComponent` method of the inherited `JPanel` by calling

  `super.paintComponent(`*whatever your Graphics object is called*`)`

- calls the `draw` method of each `Diner` object, passing the `Graphics` object as a parameter
- draws the rectangle representing the table

5. The TablePanel class will need to import `java.awt.*` in order to use `Color` and `Graphics`. It will also need to import `javax.swing.*` in order to use `JPanel` methods.

This class is complete. You might like to compile it to check for errors.

The Application class ***(compare with the Splat class L,D&C Appendix F Listing F.3 page 973)***

1. The application class will need a `main` method which

- creates a new `JFrame` object

- adds a new instance of the `TablePanel` class to it

- calls the `setDefaultCloseOperation`, `pack` and `setVisible` methods

- imports `javax.swing.*` in order to use `JFrame` methods.

Note that Java, and COMP160 lectures, have moved to using WindowConstants.EXIT_ON_CLOSE rather than JFrame.EXIT_ON_CLOSE as listed in the textbook.

Lab 14 Completed

The code I present for my Lab 14 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments
☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 15 Arrays

Notes

Readings: L,D&C Chapter 7, Sections 7.1 & 7.2 (pages 340 -351).
The power of loops can be realised when data is stored in a systematic way. An array is the first data structure we will use.
This lab uses both `for` and `foreach` loops, loops with sentinel values, application and support classes, `void` and `return` methods, methods with and without parameters, constructors, arrays, random number generators and boolean conditions. If you are able to write and understand the code for this task, you have an excellent grasp of the concepts presented in the first half of the course.

Preparation

1. Declare an array of `int` called `scores` that can hold 10 values.

2. Declare an array of `String` called `words` that can hold 5 words.

3. Declare an array of `double` called `heights` which can hold 20 values.

4. Declare an array of `String` called `family` and use an **initialiser list** to fill it with the names of your family members.

5. Write a statement which prints out the `length` of the array called f`amily`.

6. Write a `foreach` loop which prints out the name of each family member in the array called `family` on a new line.

7. Write a `for` loop which prints out the name of each family member in the array called `family` on a new line.

Plans for Lab Work

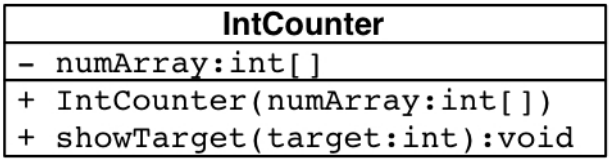
Check each method does what it should as you write it. Use `System.out.println` values freely to see whether things are as they should be e.g the random values are being created correctly. You might want to use a hard-coded array to get the rest of the code operational before writing the `makeArray` method.

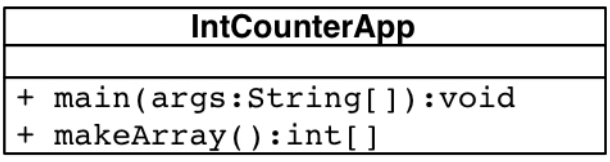
Description summary of the Support class

data field	**array of int** (an array of int is of type **`int[]`**)
constructor	**takes an array of int as a parameter** (an array of int is of type **`int[]`**)
	sets the value of the data field in the usual way
	displays every element of the array (foreach) and the length of the array on 1 line
showTarget method	**takes an int as a parameter**
	produces a display line each time the target integer is found in the array, stating in which index/position the target is found

Description summary of the Application class

makeArray method	**creates an array of int of random size between 5 and 10**
	fills the array with random values between 0 and 4
	returns the array
main method	**in a loop, asks the user for the target integer (3 times)**
	makes an instance of the support class, sending it an array of int returned by makeArray
	calls the showTarget method of the instance, sending it the target integer





Refer to Lecture 14, topic `this`, for how to successfully use the same name for a data field and constructor parameter.

Lab Work

The purpose of the program you will write for this lab is to display the array position of every occurrence of a 'target' integer that is stored in an array of `int`. We are asking you to do this with a very specific class structure, which is described carefully below, and summarised on the preceding page.

The support class will contain a single **data field** of type `array of int`. The support class will have a **constructor** which is sent a reference to an `array of int` as a parameter, and uses this to set the value of the data field. A foreach loop in the constructor will display all the elements stored in the array on one line, followed by the length of the array (see Figure 16.1). The support class also has a `showTarget` **method** which is sent a target integer as input and uses a `for` loop to access every position of the array in turn, and print out the position number (index) of the positions which hold the target value. (This target integer will be read in from the user, using a `Scanner` object in the `main` method, then passed to the `showTarget` method in the support class.)

The **application** class has a **method** which returns an `array of int`. This method uses the random number generator to decide how big the array is going to be (between 5 and 10 elements), fills it with random `int` values (between 0 and 4), then returns the array.

The **`main` method** contains code in a loop which asks the user to specify the target integer then makes an instance of the support class, sending it a reference to an array returned by the `makeArray` method. It will then call the `showTarget` method (passing it the target value). The loop will finish after three iterations.

Hints:

Use **`"\t"`** to indent the output by 1 tab stop.

An array of `int` is of type **`int[]`**

```
Which number do you wish to find?
[2]
4 4 1 2 0 0 2 4 Array is of length 8
   There is a 2 in position 3
   There is a 2 in position 6

Which number do you wish to find?
[3]
0 2 3 3 1 2 0 Array is of length 7
   There is a 3 in position 2
   There is a 3 in position 3

Which number do you wish to find?
[3]
2 0 4 2 4 2 1 4 0 Array is of length 9


Finished
```

Figure 16.1

If you have the time and the inclination, see if you can add a statement that reports that the target integer isn't found when appropriate e.g.

```
"There are no 3's in this array"
```

Lab 15 Completed

The code I present for my Lab 15 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments ☐ working
☐ structure as described ☐ submitted

Date **Demonstrator's Initials**

Laboratory 16 Two-Dimensional Arrays

Notes

Arrays of objects and two-dimensional arrays.

Readings: L,D&C Chapter 7, Sections 7.3, 7.4, 7.5, 7.6 (pages 351- 370).

1. Draw the data structure which is created by this code, and the values that are stored in it:

```
int cols = 4;
int rows = 5;
int [] [] table = new int[rows][cols];
for (int row = 0; row < rows; row++){
   for (int col = 0; col < cols; col++){
        table[row][col] = row * col;
   }
}
```

2. Using a nested for-each loop, write code to print out the values stored in `table` (Q1 above), with a tab between each column in the row, and each row on a new line.

3. Write a nested loop which produces the output (Note this does not require an array)

```
1
22
333
4444
55555
666666
7777777
```

4. Consider the `Tester` class below.

```
public class Tester{
    int[] dunedinTemps = {15, 12, 9, 13, 14};
    int[] aucklandTemps = {17, 15, 18, 14, 17};
    int[][]temps = {dunedinTemps, aucklandTemps};

     public void show(int row, int col){
        System.out.println( temps[row][col]);
     }
}
```

What would the output be if you used the method call `show(0,0);`

What would the output be if you used the method call `show(1,2);`

Write a loop which displays just the first temperature of the list for each city.

Lab Work

Part 1

The array for this exercise is an array of `String` objects which is filled by the user via a `Scanner` object.

1. In the `main` method of a new class, declare an array of `String` called `fruits` that can hold 3 `String` objects.

2. Write a foreach loop to print out all the elements in the array. Compile and run. You will see the elements are all null at this point. Unlike arrays of primitive types which are instantiated to 0 or empty, arrays of reference types need to be instantiated.

3. Between the declaration and the foreach loop, use a loop to fill each array position with the name of a fruit. Use a `Scanner` object to get a line of text from the user. Don't forget to prompt the user with a message to explain what is expected of him/her.

4. If you compile and run at this point you should be able to enter three strings then see them displayed in a list.

5. Change the statement in the foreach loop so it prints out `"Guess what fruit I am?"` , displaying the first 2 letters of each fruit and the number of letters in the fruit's name. (Just treat a space as another letter in the name for fruits such as passion fruit.)

e.g. `Guess what fruit I am?  pe  4 letters`

6. Get the guess from the user. If it is correct, print `Correct` and carry on to the next fruit. It it is not correct, print `Try again` and repeat the question.

Part 2

Write a program which displays a multiplication table up to 12 x 12 as in Figure 17.1.

Note `"\t"` will print a tab for spacing the output.

Just use an application class, and put all your code in the `main` method.

1. Declare a two-dimensional array **just big enough** to store the information, and use a nested **for** loop to fill it with the data. (Make sure you are saving `1*1` in position `0,0`.)

2. Use a nested **foreach** loop to display the data you have saved in the array.

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108
10	20	30	40	50	60	70	80	90	100	110	120
11	22	33	44	55	66	77	88	99	110	121	132
12	24	36	48	60	72	84	96	108	120	132	144

Figure 17.1

Part 3

```
public class Average{

    public static void main(String[]args){

        int [] [] table = {{1,2,3},{4,5,6},{7,8}};



    }

}
```

Given the starting code above, finish the main method so that it calculates and displays the average for each "row" of the 2 dimensional array. Each item in the row, and the average, should be displayed on one line. The output should look like this:

```
Output
1 2 3    Average : 2.0
4 5 6    Average : 5.0
7 8    Average : 7.5
```

Lab 16 Completed

The code I present for my Lab 16 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments ☐ Part 1 Fruit
☐ Part 2 12 x 12, 2 nested loops ☐ Part 3 Average ☐ submitted

Date **Demonstrator's Initials**

Laboratory 17 Options

You may choose whether to get your mark for this lab from discussing your corrections to your mid-semester exam with a demonstrator OR working through the Command Line Interface work described below.

Notes

This lab has you creating a Java program using "old fashioned" tools based on a "command line" interface in a simple terminal / console. The point of this is to be aware that there are different ways of writing and running programs. Unix commands are issued by typing appropriate characters and pressing the **<return>** (or **<enter>**) key. In this lab we give you some of the basics to get you started.

Preparation

There are NO preparation questions for this lab. We ask you instead to look at your mid-semester exam paper and improve your answers for questions you did not get full marks for.

Lab Work

Terminal / Console vs GUI

In the "good old days" a computer screen ("terminal" or "console") could only display lines of text, and respond to text commands that were typed in - a command line interface (CLI). Operating systems such as Unix and DOS (on the PC) were all based on CLIs. With the invention of the mouse and good graphics, the use of a graphical user interface (GUI) has become much more popular. The Macintosh was the first popular computer to use a GUI in 1984, and PCs eventually followed with early versions of Windows in the 1990's.

Although GUIs have more or less taken over, many current operating systems still provide a CLI as well, often in a special terminal / console window running within the GUI. See for example modern versions of Unix / Linux, including MacOS X (which has a Terminal application in /Applications/Utilities).

Programs have changed too. Older style programs were text based, reading inputs from and writing outputs to the terminal / console.

Modern programs use GUI elements with dialog boxes, windows, text fields, buttons and so on. Java allows us to do either. The programs we write in DrJava can use the DrJava console tools for text input and output, or create and use GUI elements like dialog boxes.

Unix Commands

1. Start the `Terminal` application from the **Applications> Utilities** directory. The window that opens represents a terminal / console (what used to be visible on a computer monitor in the days before graphical user interfaces and mice). You will see

```
oucsXXX:~ yourShortName$
```

The `$` is a "prompt" after which you may type a Unix command. You are now using a Unix command line interface (CLI). The path of your current working directory is shown at this prompt. In this case you are at the top level. The ~ symbol is called the **tilde** and can be found on the top left of your keyboard. It represents the top level of your **home** directory.

2. Type `pwd` (short for "print working directory") and press <return>. The complete path from the top level of the server to your **home** directory should now be shown. The top level of the machine is represented by the first `/` (back-slash) symbol.

3. Type `ls` (short for "list" – Unix people don't like typing long commands!) and press <return>. You will see a listing of all the directories and files in your **home** directory. By default when you open Terminal you are looking at your **home** directory.

4. Type `ls -al` and you will see more information about these files and directories, including size, ownership and date modified.
The characters at the beginning of each line e.g. `drwxr-x--x` tell you

a) whether this is a directory or a file (leading `d` means directory, leading `-` means file).

b) the reading, writing and executing permission for the owner - you (`rwx`), your group (`r-x`) and other users not in your group (`--x`). (Please don't try to change these permissions.)

5. Suppose you want to see what is in your **COMP160> Lab16** directory (assuming you have one).

> If you have a space in your directory name (e.g. Lab 16) type cd Lab\ 16 rather than cd Lab 16. The backslash tells unix that the next character has no special meaning. The space therefore will not be interpreted as a delimiter, separating instructions and arguments from each other, but is part of the string of characters.

Type `cd COMP160/Lab16` <return> (short for "change directory").
Now you have made the Lab16 directory your current working directory. Type `ls` again, and you will see a listing of the directories and files within it. See how the prompt changes to reflect your current directory

```
oucsXXX:~/COMP160/Lab16 yourShortName$
```

Type `pwd` again. **Note**: to repeat a command you have already used, you can use the up and down arrow (as for the Interactions Pane).

To move to your **home** directory from anywhere, type `cd ~` (the tilda can be found on the top left of your keyboard).

To move to your **Lab16** directory (assuming one exists) from anywhere, type `cd ~/COMP160/Lab16`. Because this gives a complete filepath, your starting point is irrelevant.

`cd ..` will move you back up one level in the structure (filepath) from wherever you are.

`cd ../Lab12` will move you back up one level then down into the **Lab12** folder, assuming one exists..

You can type the first letter or two of a filepath then press the <tab> key - if the letters uniquely identify a file or folder, unix will fill in the rest for you. If there is a choice, it will fill in as much as it can and wait for you to choose the next letter(s). Note that Unix, like Java, is case sensitive: desktop is not the same as Desktop.

6. Change directory back to your **COMP160** directory. Make a new directory called **Lab17** here using the command `mkdir Lab17`. Open your **COMP160** folder from the Mac OS 10 GUI and you will see a folder/directory there called **Lab17**.

7. Let's create a copy of your **Lab2> TwoNumbersApp.java** file to your **Lab17** directory. Type

```
cp ~/COMP160/Lab2/TwoNumbersApp.java ~/COMP160/Lab17/TwoNumbersApp.java
```

OR

```
cp ~/COMP160/Lab2/TwoNumbersApp.java ~/COMP160/Lab17
```

These commands contain two complete filepaths, so it doesn't matter where your current working directory is. The first version gives you the opportunity to change the name of your file. The second version assumes you don't want to change the name of your file.

If your current working directory is **Lab17**, you could type a simpler instruction:

```
cp ~/COMP160/Lab2/TwoNumbersApp.java TwoNumbersApp.java
```

OR

```
cp ~/COMP160/Lab2/TwoNumbersApp.java .
```

(the dot is short for current working directory)

or if your current working directory is **Lab2**, you could type:

```
cp TwoNumbersApp.java ~/COMP160/Lab17/TwoNumbersApp.java
```

OR

```
cp TwoNumbersApp.java ~/COMP160/Lab17
```

Don't forget to use the <tab> key for speed and efficiency.

If you wish to move a file rather than copy it, use the `mv` command. You can rename a file by moving it within the current directory like this: `mv oldfilename newfilename`

If you wish to delete a file or directory you use `rm` (remove) or `rmdir` (remove directory). Be warned - there is no Trash / Recycle Bin half-way house. When your file is removed, it is gone.

```
rm filename
rmdir directoryname
```

Java

1. With **`Lab17`** as your current directory, type `javac TwoNumbersApp.java`

This will compile the code from `TwoNumbersApp.java` and produce a class file called `TwoNumbersApp.class`. (Take a look with `ls`).

2. Type `java TwoNumbersApp` - this will run your java class called `TwoNumbersApp.class`. You should see the output listed in the Terminal window.

Nano

1. Now let's create and run a java program with the basic command line tools. Type `nano`.

This is an editor that can be used in terminal. The mouse doesn't work here (except for cut & paste). You move the cursor around with the arrow keys. There is a menu at the bottom of the window which gives you commands for opening (ReadFile) and saving (WriteOut) a file. The ^ stands for the <control> key.

2. In `nano`, type the code for `HelloWorld` from Lab 1. Save your code with `Control O`, giving it the usual file name, and exit from `nano` with `Control X`.

3. Compile and Run your program from the Terminal.

These basic commands are only a small introduction into the world of Unix commands. There is an online manual that explains commands in more detail: type `man` then the command to access this, e.g. `man ls`.

Basic Unix commands

ls = list	cd = change directory
mkdir = make directory	cp = copy
mv = move	rm = remove
man = manual	rmdir = remove directory

If you want to break out of any Unix process i.e. if you want your prompt back, type <control> c (together).

To quit from the `man` pages, press `q`.

Javadoc

In `Terminal`, `cd` to a java support class you have had marked recently e.g. Lab14's `Diner.java` and type

```
javadoc yourFileName.java
```

In the Mac OS GUI, navigate to the file's directory and double-click on `index.html` - you have now made your own java documentation.

Sending data to a program via Command Line Arguments (`String[] args`)

Write a java **main** method for a class **Argument** with the **foreach** loop shown in the box to the right in the method body. **Compile** the code to make the class file.

```
for (String s: args){
 System.out.println(s);
}
```

In DrJava's **Interactions** panel, type `run Argument one two 'three fish' four`

In **Terminal**, type `java Argument one two 'three fish' four`

Lab 17 Completed

☐ CLI tasks performed OR ☐ mid-semester exam corrections discussed

Date **Demonstrator's Initials**

Laboratory 18 Graphical User Interfaces

Notes

Readings: L,D&C Chapter 6, Section 6.1 (pages 246 – 256). Section 6.2, Text Fields (pages 257 – 260)

Preparation

1. Fill in the underlined gaps in the following code so that the inner class is instantiated correctly and the event-handling method and the reset button will function.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonPanel extends JPanel{
    JTextField display = new JTextField(12);
    JButton change = new JButton("Change team");
    JButton reset = new JButton("Reset");
    int next = 0;
    String [] teams = {"Highlanders","Crusaders", "Waratahs", "Chiefs"};

public ButtonPanel(){
        ButtonListener bl = new ____________________();
        change.addActionListener(bl);
        reset.addActionListener(bl);
        display.setText("Press a button");
        add(change);
        add(reset);
        add(display);
    }

    private class ButtonListener implements ActionListener{

       public void ____________________(ActionEvent _____________){
            if (event.getSource() == change){ // display next team
                display.setText(teams[next]);
                next = (next < teams.length-1)? ++next : 0;
            } else if (event.getSource() == _____________){   // reset display
                display.setText("Press a button");
                next = 0;
            }
       }
    }
}
```

2. What three kinds of objects are needed for user interaction? Give an example of each with reference to the Q1 example.

i)

ii)

iii)

3. Which of the following classes belong to the `java.lang` package, so are imported automatically? (Look up the Java API using the instructions in this workbook page 21)

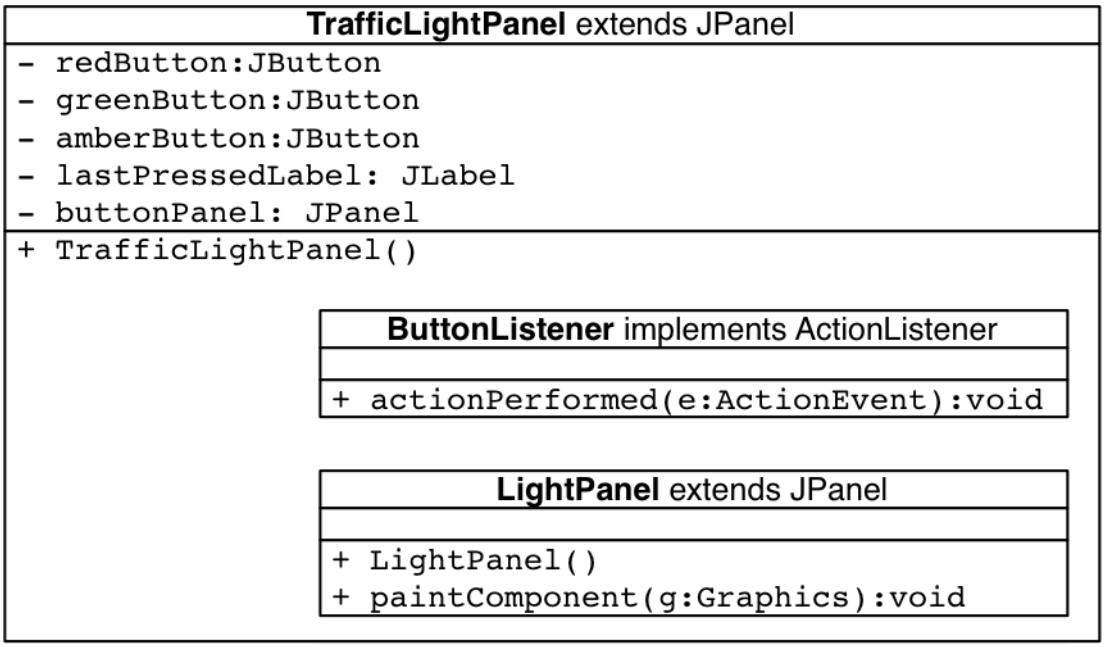
`Color`	`Graphics`
`JFrame`	`Math`
`Object`	`Scanner`
`String`	`System`
`Double`	`Random`
`JPanel`	`Integer`

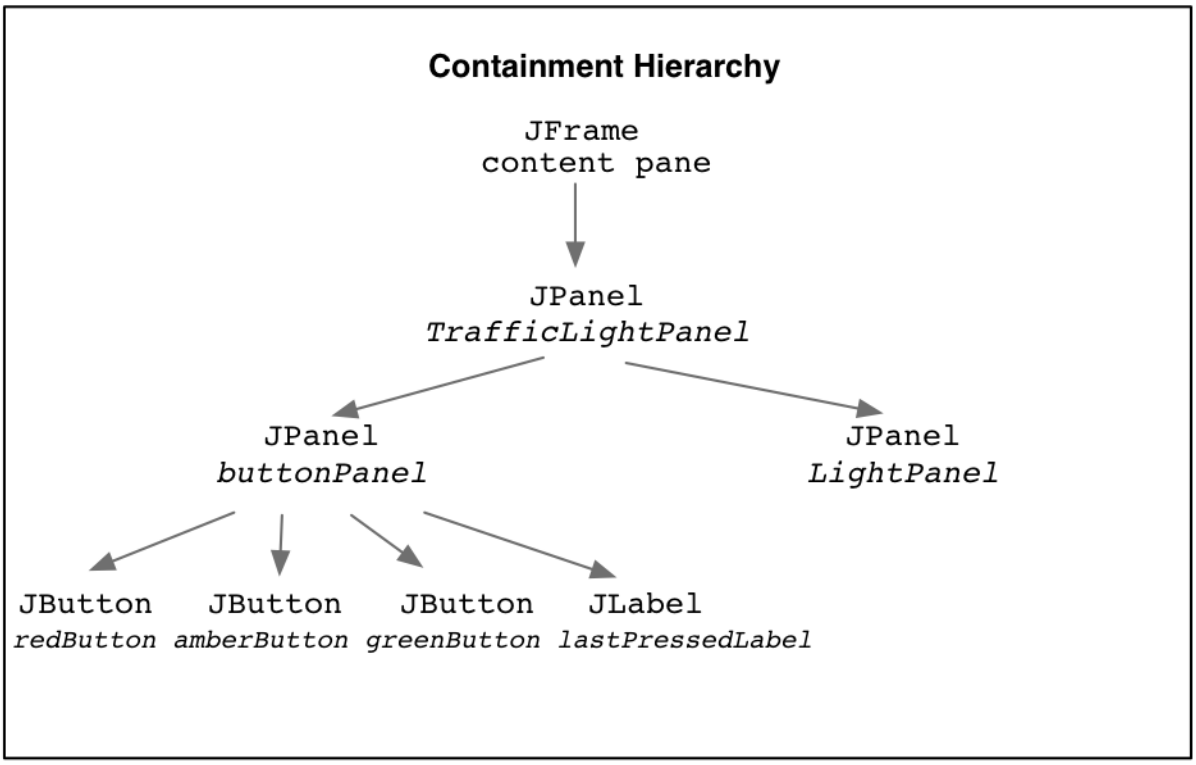
4. What method **must** you write if you `implement ActionListener`?

5. In an `actionPerformed` method with argument `ActionEvent aE`, what does `aE.getSource()` return?

6. Take a look at code listing 6.6 (L,D&C page 258-9), with particular reference to the `getText` and `setText` methods for text fields.

Plans for Lab Work





Lab Work

1. Using listing 6.3 (L,D&C page 253) , or Lab 14 DinerApp (COMP160 Lab book page 71) as a guide, write an application class which makes a `JFrame` containing an instance of a class called `TrafficLightPanel`.

2. Using listing 6.4 (L,D&C page 255) and the UML diagram on the previous page as a guide write a `TrafficLightPanel` class which:

- extends `JPanel`
- imports the 3 packages necessary for GUIs, graphics and events
- has 3 `JButton` data fields (which will show "Red" , "Amber" and "Green")
- has 1 `JLabel` data field (which will initially show the text "last pressed")
- has 1 `JPanel` data field called `buttonPanel`
- has a constructor which:
  - ♦ instantiates any data fields that still don't exist
  - * sets the size of the `TrafficLightPanel` to 200 by 300
  - * sets the background colour of `TrafficLightPanel` to blue.
  - ♦ sets `buttonPanel's` preferred size to 80 by 290
  - ♦ sets `buttonPanel's` background to white
  - ♦ adds the buttons and label to `buttonPanel`
  - * adds the buttonPanel to `TrafficLightPanel`

** in the TrafficLightPanel constructor, any instruction to apply to the TrafficLightPanel itself does not need to use dot notation.*

3. Compile and run. You should see something like the frame shown in Figure 18.1.

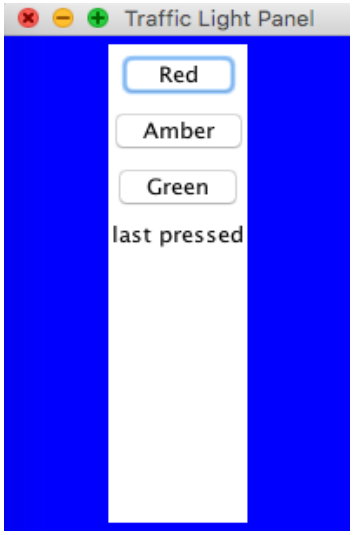


Figure 18.1

Those buttons are no use unless they do something.

Time for action!

4. Still using listing 6.4 as a guide, write an inner private `ButtonListener` class which `implements ActionListener` and has an `actionPerformed` method (it can be empty for now). In the `TrafficLightPanel` constructor make an instance of this class. Register each button to this listener object.

5. In the `actionPerformed` method, if the source of the event is `redButton`, set the text on `lastPressedLabel` to "red" and set the background colour of `buttonPanel` to red.

6. Repeat this for the other buttons (use `Color.orange` for amber).

7. Compile and run. Your frame will look the same initially, but now should respond to the press of each button.

Graphics on the JPanel

Now we want to draw a graphical representation of a traffic light and have the lights changing colour rather than the background. We could do it in the `TrafficLightPanel`, but it's not the ideal solution. Step 8 will illustrate why.

8. Write a `paintComponent` method in the `TrafficLightPanel` class. Don't forget it needs a `Graphics` object as a parameter (refer back to L,D&C page 972 Listing F.2, `SnowmanPanel`). In this method, set the graphics colour to red and draw a filled circle at `(0,30,40,40)`. Compile and run. Now change the position of the circle to `(40,30,40,40)`.

If your dimensions are the same as those suggested above, the circle will disappear behind the button panel because it is being drawn on the `TrafficLightPanel`. A better solution is to make another panel to put the lights on (Step 9). This panel will sit neatly beside `buttonPanel`.

9. Make another inner private class called `LightPanel` which `extends JPanel`. (This means it will have all the functionality of a `JPanel` as well as anything else we care to add on top, just like `TrafficLightPanel`.)

10. Shift the `paintComponent` method into the `LightPanel` class. Make the first statement of the method a call to `super.paintComponent(page);` (assuming `page` is the name of your `Graphics` parameter)

This draws the parent :- the JPanel which LightPanel extends. This will give you a background colour.

11. In a constructor for `LightPanel`, set the preferred dimension to 80 by 290 and set the background colour to cyan.

12. In the `TrafficLightPanel` constructor, make an instance of `LightPanel` and `add` it.

13. Compile and run.

14. The red circle will now be drawn with reference to the `LightPanel`, so you may need to readjust the `x` position. Now draw the other 2 circles. Your frame should look something like that in Figure 18.2.

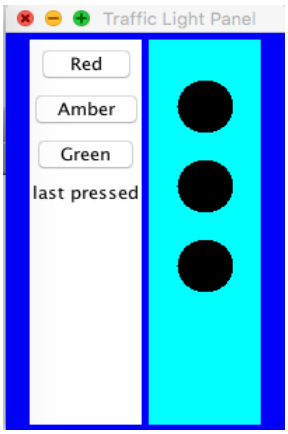


Figure 18.2

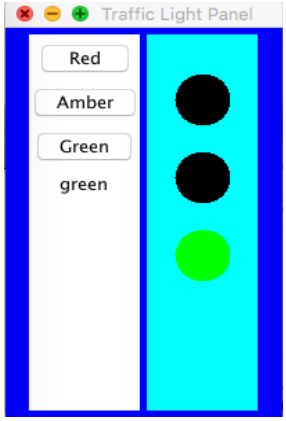


Figure 18.3

15. The final task is to make your "lights" change colour with the buttons rather than the background. Call `repaint()` at the end of the `actionPerformed` method (this refreshes the panel).

In the `paintComponent` method, start by drawing all the circles in black. Then **if** `lastPressedLabel` shows the text "red", draw the top circle again this time in red, etc. (Use the `getText` method of the `JTextField` / `JTextComponent` class - see L,D&C Section 6.2 Listing 6.6)

16. The `lastPressed` JLabel doesn't need to be added to the panel in order to function. Once your program is working and you no longer need to check what is going on, you may remove it from `buttonPanel`.

Lab 18 Completed

The code I present for my Lab 18 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments
☐ sensible variable names ☐ working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 19 Calculator

Notes

Readings: L,D&C Chapter 6, Section 6.2 & 6.3 (L,D&C pages 256 – 296), Chapter 2, Figure 2.2 (L,D&C page 47).
The Java `awt` class has an `event` package which contains classes required for user interaction. In a GUI, a user's choice can be defined by `JButtons`, `JRadioButtons` and `JCheckboxes` (among other `JComponents`).

This lab is based on a very good strategy for designing GUIs, where the system of interest (e.g. a calculator) is kept separate from the GUI that forms the "front end" (instead of, for example, trying to mix up both aspects of the program into one class). Otherwise, there are no new concepts introduced in this lab.

Preparation

1. Take a look at the code listing for the Calculator class on page 89 of this book, and in the lab files .

Which method takes an `int` as a parameter?

Which method is going to calculate a result?

Which method is going to reset all stored values?

Which method should be called if an operator key is pressed?

2. Look up `JTextField` in the Java APIs. What is the 10 in the statement below referring to?

```
private JTextField display = new JTextField(10);
```

3. Can a `JTextField` display a value of type `int`?

4. What kind of event is generated by a `JCheckBox`?

5. What is the default layout for the class `JPanel`?

6. Explain why the JRadioButtons in `QuoteOptionsPanel` (L,D&C Listing 6.10 page 263-7) are in a `ButtonGroup`.

7. Explain when you should use the data type `long`.

Lab Work

Part 1

Take the `StyleOptionsPanel` and `StyleOptionsApp` classes from the **coursefiles160> Lab19** folder. Add three functioning `JRadioButtons` which will offer three different typefaces.

Use a `GridLayout` to put all the items on the panel in a single column.

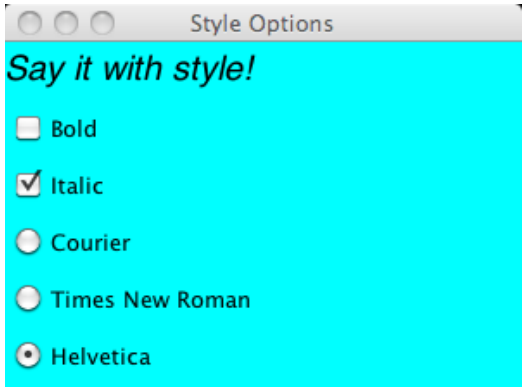


Figure 19.1

Calculator class code listing. You will need to understand what the methods do.

```
/** Calculator.java Lab 19, COMP160, 2019
 * A calculator class - for SIMPLE calculations like 5 + 20 =
 * Large inputs will overload int, should convert to long
 */
public class Calculator {

   private int currentInput;                //current input
   private int previousInput;               // previous input
   private int result;                      // result of calculation
   private String lastOperator = "";        // keeps track of the last operator entered

   /** New digit entered as integer value i - moves currentInput 1 decimal place to the left and adds i in "one's column"
    @param i a digit */
   public void inDigit(int i) {
     currentInput = (currentInput * 10) + i;
   }

   /** Operator entered + - or *
    @param op an operator symbol, one of + - * */
   public void inOperator(String op) {
     previousInput = currentInput;   // save the new input as previous to get ready for next input
     currentInput = 0;
     lastOperator = op;              // remember which operator was entered
   }

   /** Equals operation sets result to previousInput + - or * currentInput (depending on lastOperator) */
   public void inEquals() {
     if (lastOperator.equals("+")) {
       result = previousInput + currentInput;
     } else if (lastOperator.equals("-")) {
       result = previousInput - currentInput;
     } else if (lastOperator.equals("*"))  {
       result = previousInput * currentInput;
     }
     lastOperator = "";              // reset last operator to "nothing"
   }

   /** Clear operation resets all stored values */
   public void inClear() {
     currentInput = 0;
     previousInput = 0;
     result = 0;
     lastOperator = "";
   }

   /** returns the current result
    @return the current result as a string */
   public String getResult() {
     return Integer.toString(result);     //converts int to String
    }

   /** returns the previous input value
    @return the previous input as a string */
   public String getPreviousInput() {
     return Integer.toString(previousInput);
   }

   /** returns the current input value
    @return the current input as a string */
   public String getCurrentInput() {
     return Integer.toString(currentInput);
   }
}
```

Part 2

In this lab we're going to build a GUI based application, a simple calculator for integer arithmetic. (It will have a few limited functions, not the full range that you are used to in a real calculator). There are several ways to approach such a design task. One very good way is to keep the functionality of the system (in this case a calculator) separate from the GUI "front end".

We will start with a `Calculator` class, and a simple text based "front end" application. Without changing the `Calculator` class at all, we will replace the front end with a GUI based application.

1. Copy the files `Calculator.java` and `CalcTextApp.java` from **coursefiles160** into your working folder. Explore them to see how they work.

> `Calculator` has four data fields: `currentInput` (to hold the current number being input), `previousInput` (to hold the previous input) , `result` to store a result and `lastOperator` (to hold a `String` representing the last operator entered).
>
> **Note** that the `Calculator` class only has `+` `-` and `*` operations, an "equals" operation and a "clear" operation. Numbers are entered a digit at a time. If `4` has been entered so far and `2` is entered next then the current value of input is set to `(4 * 10) + 2`, i.e. `42`.
>
> The `CalcTextApp` application front end to `Calculator` is very simple. It makes a `Calculator` object, and codes a single simple calculation, `50 - 26 =`.

2. Compile and run the `CalcTextApp` application. Explore a few more calculations. Calculator can only handle calculations of the form "number operator number =". Other sequences will do strange things!

Time to build a more user-friendly GUI front end. We will do this by modifying an existing example.

3. Copy the file `CalculatorPanel.java` and `CalculatorGUIApp.java` from **coursefiles160** into your working folder. Compile and run.

We need to make a few initial changes to `CalculatorPanel`.

4. Modify the code so that when it runs it looks like Figure 19.2 (the operator keys are `=` `+` `-` and `*`).

When run, it should still behave exactly as before, e.g. the picture shows the panel after the `2` button has been pressed. (How many rows are in the grid now?)

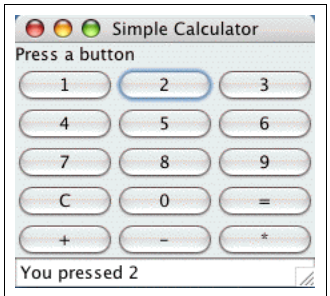


Figure 19.2

5. Now we need to connect the GUI to `Calculator`. `CalculatorPanel` needs a data field `calc` holding a (reference to) a new `Calculator` object.

6. All of the rest of the work happens in the `actionPerformed()` method, making button presses call methods on `calc`. Use "`if`" to work out what button has been pressed, and deal with each one appropriately (the partial example below shows one approach). Think about how and where to display the result. You will need to refer to the listing of `Calculator.java` on the previous page.

```
if ("+".equals(whichButton.getText())) {
   calc.inOperator("+");
} else if
...          // Stuff to add!
} else {     // if the button pressed hasn't been taken care of already, then the button must be a digit
   int i = 0;
   Scanner scan = new Scanner(whichButton.getText());
   i = scan.nextInt();
   calc.inDigit(i);
   display.setText(calc.getCurrentInput());
}
```

Note that all values displayed in the GUI should be values read from the `Calculator` object `calc` (via the accessor methods), not values created within `CalculatorPanel`. The GUI is supposed to reflect data in `calc` only. If we display values created elsewhere then they may be inconsistent with the actual state of `calc`.

7. Complete and test the application so that the calculator works correctly for sums like "234 – 72 =". We now have two different front ends for the same `Calculator` class, a GUI application and a text application!

8. If you enter very long numbers `Calculator` will break (notice that a long number suddenly becomes an odd negative number). This is because `int` only has a limited range. Convert the appropriate `int` to `long` to handle bigger numbers. Note that the conversions `Integer.toString` will need to be changed to `Long.toString`.

`Calculator` can only handle calculations of one fixed simple form. If you finished early, modify `Calculator` to correctly process input sequences like:

```
6 =

8 + 2 + 4 =

3 + 10 =  –  4 =

–6 + 9 =
```

and so on…

Getting the logic of a real calculator correct is very tricky!

Lab 19 Completed

The code I present for my Lab 19 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments altered to suit ☐ Part 1 (radio buttons)

☐ Part 2 clear, equals working ☐ digits, operations working ☐ submitted

Date **Demonstrator's Initials**

Laboratory 20 Reading from Files

Notes

Readings: - L,D&C Chapter 5.5 (pages 199 – 203), L,D&C Chapter 10 (pages 442 – 459). You should also read the sections "Tests you should make with try ..catch" (page 124) and "Java Input and Output (I/O)" (page 125) from the Readings at the back of this book.

This lab covers a common task - reading data from a file. There are a few potential hazards which need to be avoided.

Preparation

```
System.out.println("Enter an integer");
Scanner scan = new Scanner(System.in);
int newInt = scan.nextInt();
```

1. What will happen if the user enters a character which can not be stored as an integer e.g. 'w' in response to the code above?

```
public static int readInt() {
     boolean success;
     int input = 0;
     do {
        success = true;
        System.out.println("Please enter an integer");
        try {
           Scanner scan = new Scanner(System.in);
           input = scan.nextInt();
        } catch (java.util.InputMismatchException e) {
           System.out.println("Unexpected input, please try again.");
           success = false;
        }
     } while (!success);
     return input;
  }
```

2. **a.** What will happen if the user accidentally enters a `'w'` in response to the code above?

b. What is the purpose of the `do while` loop?

c. Why do the `boolean` and `int` variables need to be declared outside of the try/catch block?

Lab Work

The purpose of this lab is to read in integer values from a line in a text file, and use these values to create and draw graphical Rectangle objects. You are given a program as a starting point which follows the structure of the Lab 14 `Diner` / `Splat` files. The program given to you doesn't read its data from a file.

Part 1

1. Copy the files `FileApp.java`, `FilePanel.java`, `Rectangle.java` and `Lab20data.txt` from the **`coursefiles160`** folder into your **`Lab20`** folder.

You have been given a working program which draws 2 rectangles on a `JPanel`. The structure is very similar to the `Splat` and `Diner` programs from lab 14. Take a minute to understand how the program works. Run it.

Your task is to adapt this program so it reads the data for creating `Rectangle` objects from a file. Each line of the file will contain the data for one Rectangle.

2. Open **`Lab20data.txt`** in `DrJava` (File Format : All Files). Look at the contents.

- The first digit of each line is for fill: 0 for `drawRect` or 1 for `fillRect`
- The second digit represents colour: 1 for `Color.red`, 2 for `Color.blue` or 3 for `Color.green`
- The next 4 digits represent the `Rectangle`'s position and size: x, y, width and height.

These 6 items contain the information the `Rectangle` constructor needs. When one line of the file has been read, item by item, a new `Rectangle` object can be instantiated and stored in the `array`. Then the next line of the file is read. And so on until the end of the file. Refer to the example from the readings page 126.

`FilePanel` is the only class you will need to alter, and most of the changes are in the constructor.

3. In order to read from a file, import `java.io.*` at the top of the program. You will be using `Scanner` too, so import that while you are at it.

4. In the `FilePanel` constructor, comment out the lines which create the `Rectangle` objects, and the count increments. Declare a `String` variable called `fileName` and set it to the name of the data file you wish to open.

Inside a `try .. catch` block, declare a `Scanner` object which will get its input from the file:

```
try{
   Scanner fileScan = new Scanner(new File( fileName));

      // any code for reading from the file goes here

} catch (FileNotFoundException e){
   System.out.println("File not found. Check file name and location.");
   System.exit(1);//exit from program if no file to read
}//catch
```

5. Inside the `try .. catch` block you can use `Scanner`'s methods (`next()`, `nextInt()`, `nextLine()` etc) to read data from the file. Try the line:

```
System.out.println(fileScan.nextInt());
```

If you run the application class, you should see the first number of the file in the Interactions pane. So far so good. You could repeat this line 15 times and see every piece of data in the file, but the program does not usually know how much data is in the file. Reading past the end of the file will cause a run-time error. It is better to check whether you have reached the end of the file and then stop.

One way to do this is to use a `while` loop to check that there is still some data left in the file before you attempt to read from it.

6. Write a `while` block around the line which reads and displays the next integer in the file:

```
while (fileScan.hasNext()){
       System.out.println(fileScan.nextInt());
}//while
```

Now that you can read the data safely, let's think about reading it in meaningful chunks, and using it to create `Rectangle` objects to store in the `array`.

7. Inside the `while` loop, read 6 `int` values from the file. These values represent all the data for one `Rectangle`. Each piece of data should be stored in a local variable e.g `int fill = fileScan.nextInt();`

8. Now that the `while` loop is reading 6 values at a time, checking that it has one more integer is not ideal. Instead, check that it has another line:

```
while (fileScan.hasNextLine()){
```

Part 2 builds in further checking, to make sure each line is of the correct pattern.

Two of the data items need to be interpreted before the `Rectangle` object can be created.

9. The first integer from each line, representing fill/draw, needs to be turned into a `boolean` true / false. Use an `if .. else` statement to set a `boolean` varaible to `true` for a 1 (fill) or `false` for a 0 (draw).

10. The second (colour) variable needs to be turned into a `Color`. Set a `Color` variable to `Color.red` for a `1`, `Color.blue` for a 2 or `Color.green` for a `3`.

11. You now have all the data required to create a `Rectangle` object. Use a statement pair similar to those that have been commented out to create a `Rectangle` and store it in the `array`, but use your variables instead of literal values. Don't forget to increment the count.

12. Compile and run your code. You should see a rectangle for each line of the file.

13. To illustrate the use of a static variable, add two **int** data fields to the `Rectangle` class. Make one of them `static`:

```
private static int totalCount;
private int thisCount;
```

In the `Rectangle` constructor, increment `totalCount` by 1 and assign to `thisCount` the value of `totalCount`.

In the `draw` method, use a `drawString` statement to display the `thisCount` of `totalCount` information at position `x,y` of each `Rectangle`.

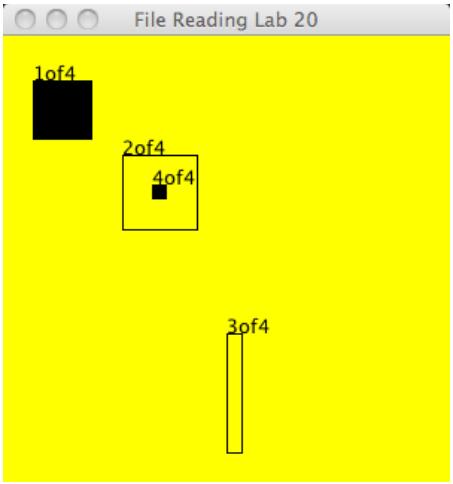


Figure 20.1

The static data field `totalCount` is shared by all instances of the Rectangle class.

Part 2

The program you wrote in Part1 has assumed the data will be 'clean' and will not make more than a particular number (10) of Rectangles. This is not always a sensible assumption for a programmer to make. There is a simple check that could be added to the code which would throw away any line of data which does not conform to the required pattern of 6 integers separated by a delimiter (space/tab).

1. First thing in the `while` loop, declare another `String` variable to hold the next input line temporarily - call it `inputLine`:

```
String inputLine = fileScan.nextLine();
```

2. To check whether this line contains the data that you are requiring, you can match the line with a delimiter pattern. (This avoids the need for an `InputMismatchException try .. catch` block.)

```
if (inputLine.matches("\\d+ \\d+ \\d+ \\d+ \\d+ \\d+")){
    //go ahead, create a new Scanner object, use it to read the 6 integers into the variables – see Step 3
}  //bad input will just be ignored - no exception handling required
```

`\\w` means a character is expected	`\\w+` means any number of characters (a word) is expected
`\\d` means one digit is expected	`\\d+` means any number of digits is expected

the **single space** between each delimiter is essential - each refers to one space " " between each token (item)

These are called regular expressions, and their documentation can be found in the Java API under `java.util.regex.Pattern` and also in L,D&C Appendix H page 1004

3. Declare and instantiate another new `Scanner` object. This time, rather than getting its input from `System.in` (the keyboard) or `new File` (the file), it wants to get its input from `inputLine`.

Use **this** `Scanner` object instead of `fileScan` to read the individual values in `inputLine`. As each value is read, store it in the appropriate variable as before. Now try your code on a data file containing corrupt data e.g. `"BadData.txt"`

4. Limit the number of Rectangles which you may attempt to store in the array – while there is another line **and** there is room for one more Rectangle in the array . . . Try this out on the file `LongBadData.txt`

Lab 20 Completed

The code I present for my Lab 20 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments ☐ Part 1
☐ static ☐ Part 2 ☐ submitted

Date **Demonstrator's Initials**

Laboratory 21 Shapes 1: Building the Structure

Notes

Readings: L,D&C Chapter 8, Sections 8.1, 8.2, 8.3 (pages 380 – 398).
This lab is the start of a sequence of four labs that build incrementally to a fairly complex final program. The sequence of labs will illustrate important OO concepts like hierarchies and abstract classes in the context of building an interactive graphical animation. We hope that you will find it both useful and fun!

In this lab you will make an array of (references to) graphical objects. This lab is drawing together all your previous knowledge of arrays, objects, events and graphics. You will be using the keywords `extends` and `super`, and may be able to develop a better understanding now of how these words help you navigate around the class hierarchy that sometimes is provided for you by the APIs, and at other times you create for yourself.

Preparation

1. The class `Art` is extended by 2 classes, `Sculpture` and `Print`. The `Sculpture` class has its call to the `super` constructor written. Complete the `Print` constructor so all its data fields are correctly instantiated.

2. If there were to be other data fields `year` and `creator`, which class should they be in?

_____

```
public class Art{
     protected int height;
     protected int width;
     public Art(int wd, int ht){
        width = wd;
        height = ht;
     }
  }

  public class Sculpture extends Art{
     private int weight;
     private int depth;
     public Sculpture(int wd, int ht, int dpth, int wt){
        super(wd, ht);
        depth = dpth;
        weight = wt;
     }
  }

  public class Print extends Art{
     private int numberMade;
     public Print( int wd, int ht, int num ){




     }
  }
```

3. There is one error in class B, and one error in class C which will stop the code below from compiling. What are the errors?

```
public class Lab21App{
  public static void main(String [] args){
    A a1 = new C(4, "Red", "Blue");
  }
}
```

```
public class A{
  protected int y = 21;

   public A (int i){
        this(i, 3);
    }

    public A (int i, int j){
        y = i / j * j;
    }
}
```

```
public class B extends A{
  private String s;

  public B(String s, int x){
    super(x * 4);
    this.s = s;
  }
  public String getS(){
    return s;
  }
}
```

```
public class C extends B{
  private int x;
  private String a;

  public C (int x, String a, String s){

    this.x = x;
    this.a = a;
    super(s, x);
    s = "Green";
    System.out.println(" x is " + x);
    System.out.println(" a is " + a);
    System.out.println(" s is " + super.s);
    System.out.println(" s is "  + getS());
    System.out.println(" s is " + s);
    System.out.println(" y is " + y);
  }
}
```

4. What will be printed out by the code if the two errors are fixed?

x is _____

a is _____

s is _____

s is _____

s is _____

y is _____

5. For each of the classes listed, write the name of the class beside the `import` statement which applies to it:

`JPanel` `JFrame` `Graphics` `FlowLayout` `Color` `Random` `ActionListener`

`import javax.swing.*` _____

`import java.awt.*` _____

`import java.awt.event.*` _____

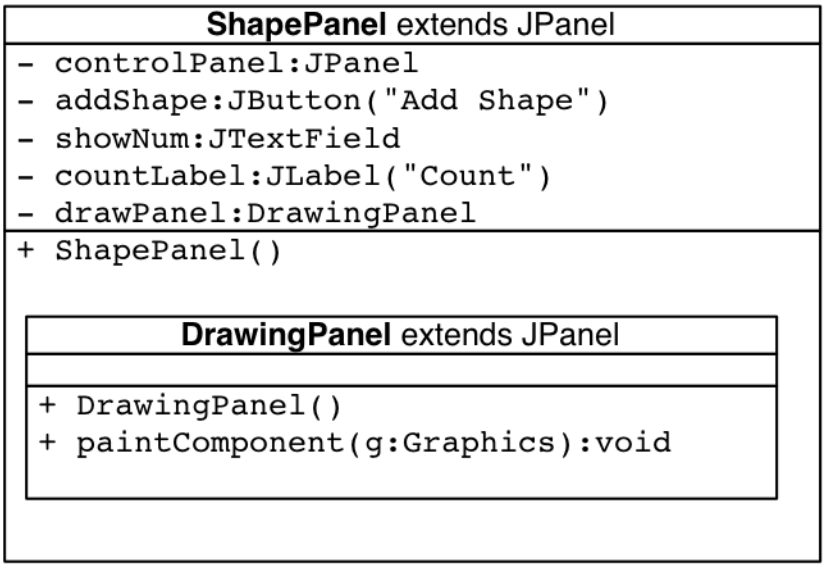
`import java.util.*` _____

6. Look up the class `Object` in the Java API. Can you explain why every class has a default `toString` method?

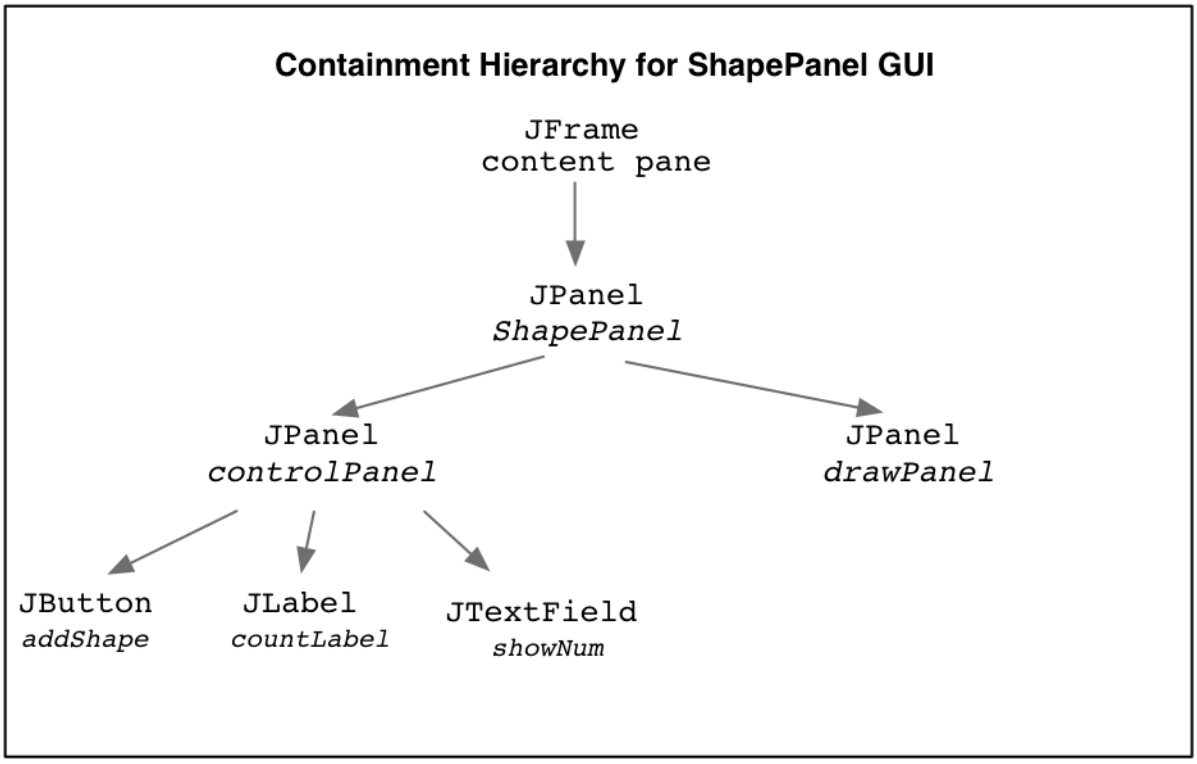
7. Remind yourself how the `Random` class works (Lab 8, page 42 this lab book or L,D&C Section 3.4 page 86)

Planning for Lab Work

Initial UML for ShapePanel class



Containment Hierarchy



Lab Work

This lab works incrementally towards its goal: a GUI program which adds a coloured circle of random size to a random location on a panel at the click of a button. Up to 20 circles may be drawn. In subsequent labs these circles will be set in motion, and shapes other than circles will be used.

The previous page describes a `ShapePanel` class and a containment hierarchy for `ShapePanel`.

- The `controlPanel` should be 100 x 400 pixels.
- The `JTextField` should have space for 2 characters.
- The `drawPanel` should be 400 x 400 pixels and have a specified background colour.
- You will also need an application class containing the `main` method which creates the `JFrame` and places an instance of `ShapePanel` on the `contentPane`.

1. Create the graphical components for the project as described. Refer back to LightPanel in Lab 18 if you need to.

Your project should produce something like the image to the right.

NOTE: If you cannot see the background colour, make sure you are calling the `JPanel`' s `paintComponent` method using `super`. (See Lab 18)

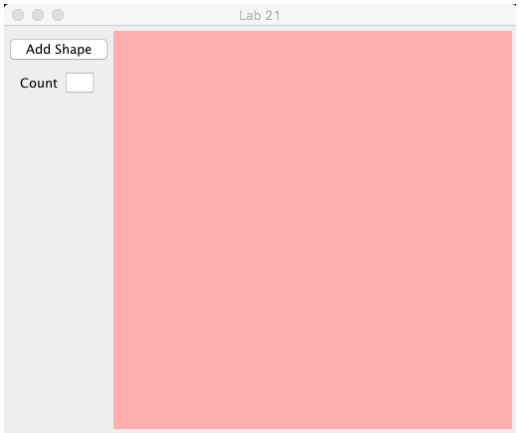


Figure 21.1 Shape window after Step 1

2. Now that the graphical structure is written, the next step is to write the `Shape` class as described by the UML diagram. The `Shape` class may remind you of the `Diner` class from lab 14. There was a `randomRange` method in Lab 8.

Shape
- x:int - y:int - width:int - height:int - colour:Color
+ Shape() + randomRange(lo:int, hi:int):int + display(g:Graphics):void

In the constructor:

- `width` is set to a random value between 10 and 30
- `height` is the same as `width`
- `x` is a random value between 0 and the width of the DrawingPanel (400) less the width of the Shape.
- `y` is a random value between 0 and the height of the DrawingPanel (400) less the height of the Shape.
- `colour` is set to a random RGB value. A new `Color` can be created using the syntax `new Color(red, green, blue)` where red, green and blue are each integer values between 0 and 255.

The `display` method should set the `Graphics` colour, and draw a solid circle using the information stored in the data fields for size and location.

3. Test your class by adding a `Shape` data field to `ShapePanel`. Don't forget to instantiate a new `Shape` object e.g. `Shape shape = new Shape()`. In the `paintComponent` method call the `display` method on your `Shape` object. Once again, refer back to Lab 14 or Lab 18 if you need to.

If things are set up correctly, each time you run the code you should see a circle of varying size in a random colour in a random position.

4. Adapt your code so that a different `Shape` will be instantiated and displayed each time the user presses the `addShape` button. This will involve registering your `JButton` as a listener and making a new Shape in the actionPerformed method (see the `ButtonListener` class in the UML diagram below, and refer back to Lab 18). Don't forget to `import java.awt.event.*` at the top of your class.

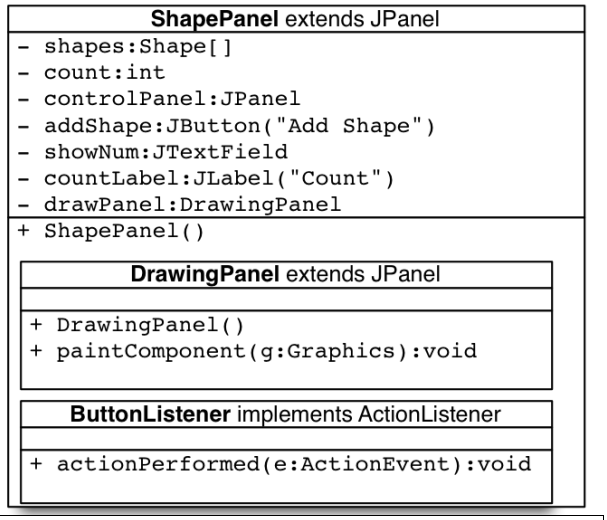
The screen will need to be refreshed each time there is a different shape to display. Use the `repaint()` method. (The `repaint` method automatically calls the `paintComponent` method, which begins by redrawing the super `JPanel` thereby removing the previous shape.)

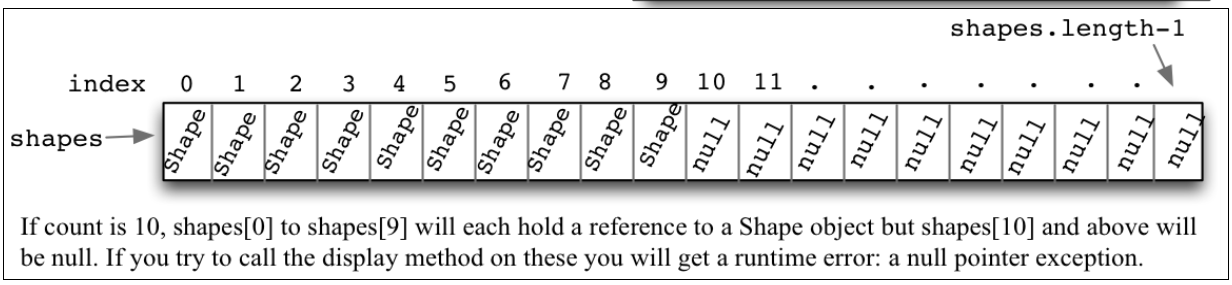
5. To complete the task, rather than have just one `Shape`, the `ShapePanel` class should have a `shapes` data field which holds an array of references to `Shape` objects. The size limit for the array will be 20.

A new `Shape` will be added each time the user presses the `addShape` button.

A `count` data field should keep track of how many `Shape` objects have been created. The current count should be displayed on the `showNum` `JTextField`.

A `for` loop in the `paintComponent` method should iterate over every **valid** `Shape` in the array and call its `display` method.





If count is 10, shapes[0] to shapes[9] will each hold a reference to a Shape object but shapes[10] and above will be null. If you try to call the display method on these you will get a runtime error: a null pointer exception.

6. Lastly, tidy up your code so that no attempt to add a shape is made when the count reaches the array length. Ensure you have written good java doc style comments (`/**` `*/`) above all method headers, as you will be working with this code for the next 3 labs.

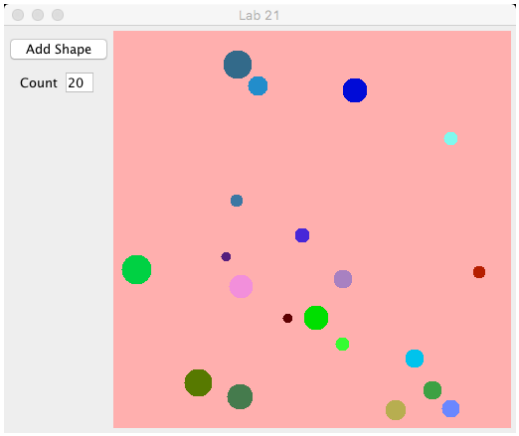


Figure 21.2 Shape window after Lab 21 Step 5

Lab 21 Completed

The code I present for my Lab 21 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ stops at 20 with no errors

☐ comments ☐ working ☐ submitted

Date **Demonstrator's Initials**

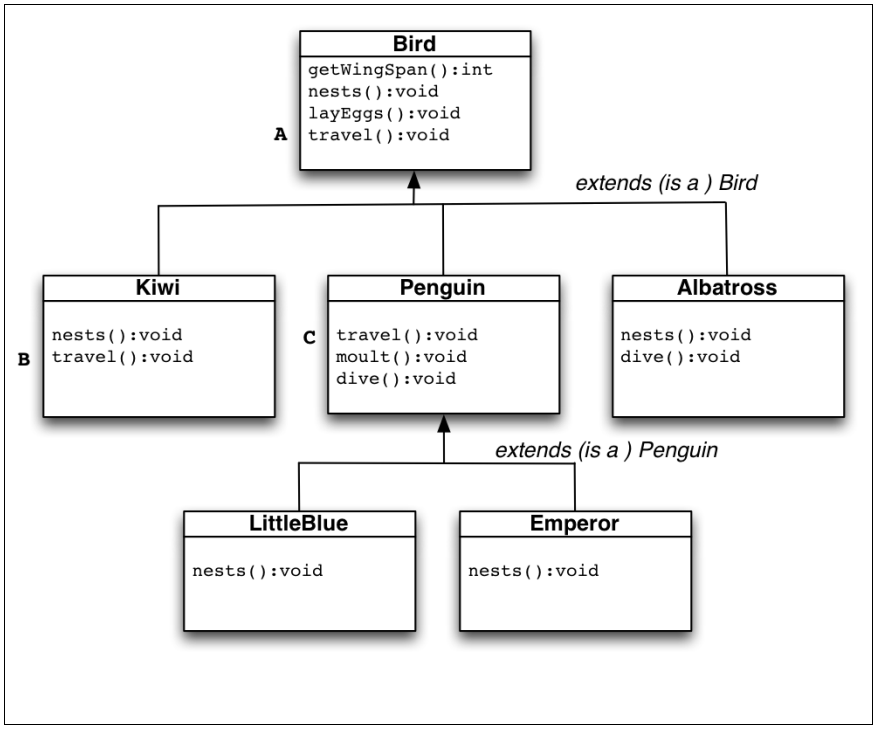
Laboratory 22 Shapes 2: Animation

Notes

Readings: L,D&C Chapter 6, Section 6.2 (pages 277 – 282). L,D&C Chapter 8, Sections 8.4 & 8.5 (pages 399 – 403). The concept of inheritance is fundamental to object oriented programming. Inheritance should be used when a behaviour is shared among classes of the same general type. The preparation exercises cover some of the concepts involved. The lab work continues the project started in Lab 21, using the `Timer` class to create action on your `DrawingPanel`.

Preparation

1. Examine the class hierarchy in the diagram, then answer the questions below.



a) There are three different `travel` methods, **A B** and **C**.

Which one does `Kiwi` use?

Which one does `LittleBlue` use?

Which one does `Albatross` use?

b) List all the methods that `Kiwi` can use?

c) Can `LittleBlue` use the `getWingSpan()` method?

d) Can `Albatross` `moult()`?

e) How can `Kiwi` access `Bird`'s `nests()` method?

f) Declare an array which may hold up to 5 elements, of type `Kiwi` **OR** `Albatross` **OR** `Penguin`.

Write another statement which makes the first element of the `array` an instance of `LittleBlue`.

2. Why is inheritance useful?

3. What is over-riding? Give an example from the `Bird` hierarchy.

4. In an `actionPerformed` method, the two statements below would each store a reference to the component that generated the event in a variable named `source`. The first casts the source to an `Object`, the second casts to a `JButton`. The `JButton` class has a `getText` method but the `Object` class does not.

```
Object source = event.getSource();              //line 1
JButton source = (JButton) event.getSource(); //line 2
```

	Yes / No
If `line 1's source` is a `JButton` would the line produce a runtime error?	
If `line 1's` source is a `JCheckBox` would the line produce a runtime error?	
If `line 1's` source is a `Timer` would the line produce a runtime error?	
Can you call `JButton's getText` method on `line 1's source?`	
`If line 2's source is a JButton` would the line produce a runtime error?	
`If line 2's` source is a `JCheckBox` would the line produce a runtime error?	
`If line 2's` source is a `Timer` would the line produce a runtime error?	
Can you call `JButton's getText` method on `line 2's source?`	

Lab Work

In this lab, you are going to get your spots moving. The `Shape` class will have a `move` method which updates the `x` and `y` (location) data fields by some formula each time it is called.

In order to get the animation working, the `ShapePanel` class will declare an instance of `Timer` which automatically generates an `ActionEvent` at regular intervals (possibly fast). This means that the `actionPerformed` method will get called at regular intervals. If we put code that updates and redraws the locations of `Shapes` inside the `actionPerformed` method, each time the `Shapes` are redrawn they will be in a different place, creating the illusion of movement (in the same way that motion pictures display the individual frames of a film).

The `Timer` class has `start` and a `stop` method which can be used to control the action. You will add a button to start them moving, and a button to stop them moving.

Putting your files in a package

1. Make a copy of your **`Lab21`** directory and rename it **`Lab22`**. Open the files in `DrJava`.
2. At the top of each class (after the comments, before the import statements) write the line

```
package shapes;
```

3. Save the files and close them. Make a new directory called **`shapes`** in your **`Lab22`** directory. Move your files into this new directory. Open them with `DrJava`.

   Your files are now in a package. Package visibility is the default visibility for Java – the visibility you get if you don't specify public, private or protected. Data fields with package visibility are available to all classes in the same package. Files in a package must be stored in a directory of the same name as the package (**`shapes`** in our case).

The Shape class

1. Declare two `int` data fields called `moveX` and `moveY` in the `Shape` class. Set their initial values to 1.
2. Write a `void` method called `move`. In this method, add `moveX` to the value of `x`, and add `moveY` to the value of `y`. This changes the location at which the shape would be drawn by 1 pixel to the right and 1 pixel down every time it is called.

The ShapePanel class

1. Declare two new `JButtons` as data fields in the `ShapePanel` class, one to `Start` and one to `Stop`. Make these buttons capable of event handling. Add them to the `controlPanel`.
2. Declare a data field of type `Timer` called `timer`, and declare a `final int` data field DELAY initialised to 10 (milliseconds).

```
Timer timer;
private final int DELAY = 10;
```

There are three `Timer` classes in the Java API. The `Timer` we are using is a class in the `javax.swing` package. If you get a multiple Timer error when using `Timer`, you can fix this by prefixing every use of the class name `Timer` with the full package name i.e. `javax.swing.Timer`

3. In the `ShapePanel` constructor, set the variable `timer` to be a new instance of `Timer`, and send it the DELAY and the `ButtonListener` as parameters.

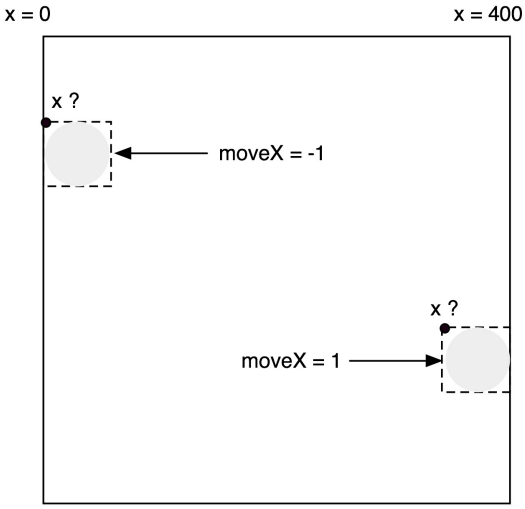
```
timer = new Timer(DELAY, listener); // or whatever you have called your ActionListener
```

4. In the `actionPerformed` method, begin with an `if` block for when the source of the event is `timer` and an `else` for when the source of the event is the `addShape` button. For the timer, each valid element in the array should have its `move` method called in turn. This will repeat the `x`,`y` updating of each `Shape` at a rate controlled by the DELAY parameter. Make sure the `repaint` is being called at the end of the `actionPerformed` method, not in an `if` block. The `repaint` will redraw the panel (every 10 milliseconds) with the `Shapes` in their new position.
5. In the `actionPerformed` method, call the method `timer.stop()` if the `stop` button is the source, and `timer.start()` if the `start` button is the source.
6. Compile and run your code.

   If all is well, your shapes will quietly slip off the right or bottom of the panel, never to be seen again. You need to keep them in view by bouncing them off the sides of the panel.

The `move` method will need to check whether the current `x` or `y` location has put the shape on the edge of the drawing panel. If it is on the edge, either the horizontal (moveX) or vertical (moveY) direction should be reversed.

How can you determine whether the shape is on the edge? Take a look at the diagram below, imagining that the shape on the left edge is travelling left (`moveX = -1`), and the shape on the right edge is travelling right (`moveX = 1`).



7. Write an `if` statement in the `move` method which checks to see whether `x` is on or over the edge of the panel. If so, change `moveX` to `-moveX`, toggling the direction of the horizontal movement. Then (whether or not you have had to change direction) carry on and make the change to `x`.

8. Write another statement to look after the vertical movement. Run your code again. Hopefully now your shapes are contained within the panel.

When you're ready, go and change some things.

- Try changing the delay to 5, then to 20. See what happens. (After each alteration, you should return the code to its original state.)
- In the `move` method, comment out the line which increments `y`. Can you predict what will happen?
- In the `Shape` class, set the value of data field `moveX` to 5.

Some challenges.

- Make shapes which are wider than 15 pixels travel straight up and down, while all other shapes travel sideways.
- Get the shapes which are initially drawn in the lower half of the screen to start their travel in an upwards rather than downwards direction. (There is a very simple solution to this one – it happens in the Constructor.)

Your mission:

- Get your shapes to change over time according to some criteria of your own design. Leave your code in this state for marking.

Lab 22 Completed

The code I present for my Lab 22 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ comments
☐ working, in a package ☐ shape change ☐ submitted

Date **Demonstrator's Initials**

Laboratory 23 Shapes 3: Abstract

Notes

Readings: L,D&C Chapter 8, Section 8.3 (pages 394 – 398) and Chapter 9, Section 9.2 (pages 413 – 425).
The text book gives an example of an abstract class in Section 9.2, the `StaffMember` class. See if you can follow how the class hierarchy works.

Preparation

1. **a.** Finish the Car and `Boat` constructors so that all data fields are correctly filled.

b. If a car's fuel consumption is calculated by mutiplying litres per kilometre by trip length, and a boat's fuel consumption is calculated by mutiplying litres per hour by trip length, write the missing (required) methods.

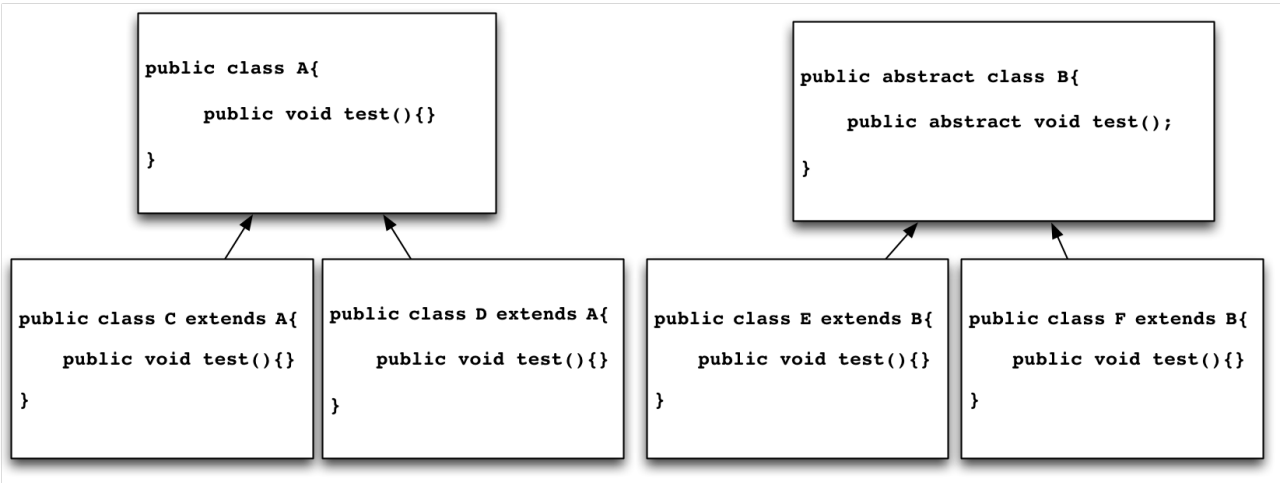
```
public abstract class Vehicle{                          //(Adapted from LDC Section 8.3)
    protected String name;
    protected String countryOfOrigin;

    public abstract int fuelConsumption(int tripLength);

    public Vehicle(String name, String countryOfOrigin){
        this.countryOfOrigin = countryOfOrigin;
        this.name = name;
    }
}
```

```
public class Car extends Vehicle{
    private int numAirBags;
    private int litresPerKilometre;

    public Car(String name, String countryOfOrigin, int lpK, int airBags){
a. ➜




    }
b. ➜ public




   }
    public String toString(){
        return "Car with " + numAirBags + " air bags made in " + countryOfOrigin;
    }
}
```

```
public class Boat extends Vehicle{
    private int litresPerHour;
    private int numberOfBerths;

    public Boat(String name, String countryOfOrigin, int lpH, int brths){
a. ➜




   }
b. ➜ public




  }
    public String toString(){
        return "Boat with " + numberOfBerths + " berths made in " + countryOfOrigin;
    }
}
```

c. In the `main` method of the application class (below), write a foreach loop which prints out what is returned by the `toString` method of every element in the `vehicles` array.

```
public class VehicleApp{
    public static void main(String[]args){
       Vehicle[] vehicles = {new Car("Toyota Starlet", "Japan", 25, 0), new  Car
         ("Volvo XC40", "Sweden", 20, 2),new Boat("Markline 700 Sport", "New Zealand", 40, 2)}
    c. ➜
```



1. Which two of the classes above could legally call a method named `test` even if you removed the method definition from that class?

2. Which one of the classes above can not be instantiated?

3. Which two classes' `test` method could legally contain the statement `super.test()` ?

4. Which two classes **must** each legally contain a method called `test`?

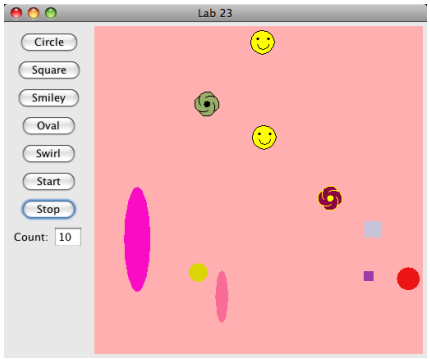


Figure 23.1 Shape window after Lab 23

Lab Work

In this lab, we are going to explore some of the possibilities arising when our `Shape` class is made `abstract`, and our `Shape` can be something more than just a circle. With `Shape` as an abstract class we can have any number of classes which extend `Shape`, each of which can represent a different sort of `Shape` (e.g. a square, an oval, or a more complicated image).

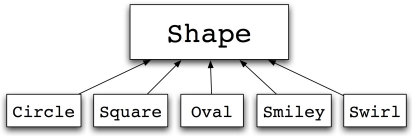


Fig 23.2 Planned Shape hierarchy

Shape to abstract

We can very quickly adapt the finished Lab 22 code to the abstract format.

To divide the `Shape` class into its abstract and non-abstract components, we need to think about the things **all** `Shape` objects will want. Since all `Shape` objects will want to be drawn somewhere, and move, they will all want the data fields `x`, `y`, `width`, `height`, `shade`, `moveX` and `moveY`.

The constructor method must remain with the abstract class. The `move` method is also common to all `Shape` objects, and can stay with the abstract class.

The method which needs to be different for each of our yet-to-be-written classes is the `display` method. At present, it draws a circle.

1. Make a copy of your **`Lab22`** folder and rename it **`Lab23`**. Open the files in `DrJava`.
2. Make `Shape` an `abstract` class. It can not now be instantiated.
3. Make a new class called `Circle` which extends `Shape` and put it in the `shapes` package. This class will also need to `import java.awt.*` in order to easily access `Graphics` methods and `Color`. (Importing is not an inherited characteristic - it just saves you from typing the full name of the class each time you use it e.g. `java.util.Scanner scan = new java.util.Scanner(System.in)`)
4. Copy the existing `display` method from `Shape` into the new `Circle` class.
5. Turn the `display` method of `Shape` into an `abstract` method (with no method body). This forces all classes that extend `Shape` to supply a `display` method.
6. Compile. If your data fields are private, you will get an error message. `Circle` needs to access the inherited data fields of `Shape (x, y, height etc.)`, so they will need to have `protected` rather than `private` visibility.
7. Compile. Another error message: the abstract class Shape cannot be instantiated. In `ShapePanel`, instead of adding a new `Shape` when the "Add Shape" button is pressed, add a new `Circle`. There is no need to change the array declaration - it still holds references to `Shape` objects. Each `Circle` is also a `Shape` because of the type hierarchy. Neat!

   Compile, fix any other errors, and run. Now you have code which will exhibit exactly the same behaviour as before, but the scene has been set for growth.

8. Write another class called `Square` which also extends `Shape`. Write a `display` method in `Square` which draws a square rather than a circle. This is all very well, but you haven't got the button structure set up yet to be able to draw a `Square`.

Array of JButtons

It's time to set up a few extra buttons on the `controlPanel`, so that you can draw your `Square` as well as some other shapes when you have written some more classes. There will be at least 7 buttons (see Figure 23.1 on the previous page), so let's consolidate the code by dealing with them in an array. Repeated tasks can be performed using a loop.

1. In `ShapePanel`, declare an array of `JButtons` that will hold 7 elements.
2. Fill the array with new `JButtons` labelled Circle, Square, Oval, Smiley, Swirl, Start and Stop.
3. In the `ShapePanel` constructor, write a `foreach` loop which accesses each element in the array and

   - adds a listener

   - adds the button to the control panel.

4. Remove all references to the "Add Shape", "Start" and "Stop" `JButtons` from the constructor.

Where the `actionPerformed` method before was using (hopefully) meaningful names for the buttons, it would now need to be referring to something like buttons[0], buttons[1] etc. Matching the right element of the button array to the right event will be prone to error. A neater solution would be to access the (meaningful) text on the button itself. The timer (which is not a button) needs to be dealt with separately.

Make the `if . . timer` block the first block in the actionPerformed method. Follow it with an `else` block which includes all the remaining code in the method EXCEPT the call to `repaint()`. This `else` block will deal just with button presses, so in it you can safely cast the object source into a `JButton` variable.

```
JButton button = (JButton) e.getSource();
```

Now each `if` statement for which button was pressed can access the text on the button itself e.g.

```
if(button.getText().equals("Circle")){
```

It is now much easier to match the event to the right process than it would have been with the code `if(e.getSource() == buttons[0]){`

5. In the `actionPerformed` method, alter the Circle, Start and Stop buttons to the pattern described above. Add the required number of `if..else` statements to cope with all the new buttons. Two of the classes to be called have been written already (`Circle` and `Square`), so test your code now using these two buttons. We need to make more classes to extend Shape . . .

More Shapes

1. Class `Oval` will look very similar to `Circle`, except it needs a constructor which sets `height` to `4 * width`. This will muck up the `y` location, which may draw the oval initially below the bottom edge of the panel, so the `Oval` constructor should also calculate a new random value for `y` using the new value of `height`.
Note: `Oval` can use the "random" method of its parent (`Shape`) class because the method has `public` visibility.

2. To make `Smiley` easier to draw, it is always going to be a standard size - 30 by 30 pixels. A `Smiley` constructor can set `height` and `width` to 30, and recalculate both `x` and `y`.

3. In `Smiley`'s `display` method, draw a filled yellow circle, an unfilled black circle, 2 black eyes and an arc mouth. To save you time try these parameters:

draw the left eye at `x + 7, y + 8, 4` pixels across.

draw the right eye at `x + 20, y + 8`

try the arc at `x + 8, y + 10, 15, 13, 190, 160`

4. A file called `Swirl.java` is in the **coursefiles160** folder.

5. Link all your button presses to calls to the correct class constructors, and off you go. Your various shapes (Circle, Square, Smiley, Oval, Swirl) can look different, but are behaving in the same way because they are all extensions of the `Shape` class.

If you want to be really clever, see if you can make `Smiley` smile on the way up and frown on the way down.

If you have still got some size changing going on from Lab 22, this may not be appropriate for Swirl and Smiley. There are three ways you could deal with this : 1) remove it 2) make `move` an `abstract` method in `Shape` and have different versions of it in each child class, or 3) in the `Shape` class, use the expression
`if (!(this instanceof Swirl) && !(this instanceof Smiley)){ //`*leave out smiley & swirl from the size changing*

Lab 23 Completed

The code I present for my Lab 23 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ foreach JButton loop
☐ comments ☐ getText to identify event source ☐ submitted

Date **Demonstrator's Initials**

Laboratory 24 Shapes 4: ArrayLists

Notes:

Readings: LDC 3.8 for wrapper classes and their use, and automatic conversion from a primitive to a wrapped object ("autoboxing"), and from an object to a corresponding primitive ("unboxing").

The `ArrayList` reading (page 130 in the Readings section at the back of this lab book).

This is the last of the Shapes labs. You are already familiar with an array, which may hold references to objects. An `ArrayList` is an ordered list of objects of flexible length. This lab illustrates graphically the dynamic nature of `ArrayLists`.

An `ArrayList` must hold references to objects. Unlike an `array`, an `ArrayList` has no fixed size. It can grow and shrink as required. Data (objects) may be inserted and removed at any position, but the process is most efficient if new objects are added to the end of the `ArrayList`.

In this lab you will convert your `array` of Shapes to an `ArrayList` of Shapes.

Preparation

1. **a)** Write a statement to declare an `ArrayList` of objects called `list`.

b) Write a statement to declare an `ArrayList` of objects called `list` which may only hold `String` objects.

c) Write a statement to add "jam" to `list`.

d) Write a statement to add "juice" to `list`.

e) Write a statement to insert "bread" as the first element (index 0) of `list`.

f) Write a statement to store the number of elements in `list` in a variable called `listSize`.

g) Write a statement to remove the element of `list` at index 1.

h) Write, in order, all of the values held in `list` after the statements above have been executed.

i) Write a statement which assigns `true` to a `boolean` variable `oj` if list contains "juice".

j) Write a statement to find out where "juice" is stored (i.e. its index), and store this value in an `int` variable `index`

2. The 2 questions below refer to an `ArrayList` called `zoo` which contains objects of type `Animal`.

```
<Animal> zoo
```

a. Write a `foreach loop` which prints out the value returned by the `toString` method of **each** element of `zoo`.

b. Write a statement which prints out a the value returned by the `toString` method of the **third** element of `zoo`.

3. An ArrayList can only hold (references to) objects and yet both of the following statements will add an object representing the integer value 3 to an ArrayList called `myList`. The first statement can be seen to be using the `Integer` wrapper class.

```
myList.add(new Integer(3));
myList.add(3);
```

What is the term for the process which is happening in the second statement?

4. The `Integer` wrapper class has many useful methods. You have seen `Integer.toString(int num)` used in order to display an `int` on a `JLabel`. There is a reverse method, `Integer.parseInt(String str)`, which converts a `String` into an `int`. Write a statement which will convert the text showing in a `JTextField` called `jt`, and store it in an `int` called `jtValue`.

5. If the text in the `JTextField jt` in Ex. 4 above could not be converted into an `int` (say it was showing an 'a'), what would happen?

Lab Work

In this lab you will convert your array of Shapes to an `ArrayList` of Shapes, then display beside each `Shape` the index number representing that `Shape` object's position in the `ArrayList`. A `Remove` button will remove a `Shape` at a chosen position. You will be able to see the `ArrayList` growing and shrinking as you add and remove `Shape` objects. NOTE on a Mac (therefore in the lab) you should use the jar version of DrJava (see Lab 1) which runs Java 8.

1. Make a copy of your Lab23 folder and rename it Lab24. Open the files in DrJava.

2. Change the declaration of the array of Shapes into an ArrayList declaration (you will need to import `java.util.ArrayList`) using the following syntax (but change *`shapes`* to whatever your array was called):

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

3. Change the syntax for adding new `Shape` objects

from `shapes[count] = new Circle();`
to `shapes.add(new Circle());` etc.

There is no longer any need to check the number of elements stored. An `ArrayList` can grow as required.

The call to the display method now needs to use `ArrayList` syntax.

4. In `Shape`, write a `showIndex` method which takes a `Graphics` object and an integer as input, and draws the integer, in black, at position `x`, `y`. (Remember the `drawString` method in Lab 14 - Diner?)

5. In `ShapePanel`, call this method in the for loop which draws each shape, directly after the call to the `display` method. Send it a `Graphics` object and the index number of that `Shape`.

6. Run your code. You should see a number beside each Shape.

7. Make a "Remove" button.

8. Change the `JLabel` so it shows "`Remove which?`" and make it display the index of the last element of the `ArrayList`.

(Remember, if `size()` is 10, the index runs from 0 to 9).

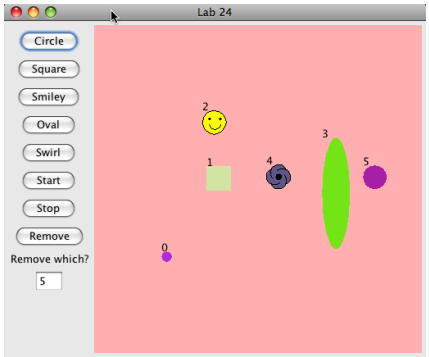


Fig 24.1 The finished Shape window

9. In the `actionPerformed` method, if the `Remove` button is pressed, store the number showing in the `JTextField` in an `int` variable. The text will need to be converted to an `int` - see preparation exercise 2. Add another statement to remove the element at this position. (This will be the last element unless the user changes it).

10. Compile and run. Add three shapes, then remove the shape at position 0. Can you see the other two shapes moving up one index position?

11. Click on the Interactions Pane then remove **all** your shapes and keep clicking the Remove button. You will see a run-time error.

12. Add a statement so that when there are no elements in the `ArrayList,` the `JTextField` shows a blank rather than `-1`.

13. Add code to catch any `NumberFormatException` when the Remove button is pressed. This would occur if there was an attempt to convert the the blank field (or any non-digit character) into an `int`.

14. One last possibility for error is that the number in the text field is greater then the size of the non-empty `ArrayList.` This would be an `IndexOutOfBoundsException` so catch that too. L,D&C Listing 10.2 page 446 gives an example of two `catch` statements being used with one `try`.

Now your code should be robust enough to cope with anything a user may type in the text field.

Lab 24 Completed

The code I present for my Lab 24 mark is my own work according to the definition on page 7 _____

☐ preparation exercises complete ☐ remove working ☐ exceptions
☐ comments ☐ "count" data field removed ☐ submitted

Date **Demonstrator's Initials**

Laboratory 25 Options

Notes

This is the last lab for COMP160! Its purpose is to set you free on some code to have some fun. Demonstrators are unable to help you with your code, so be prepared to use the APIs and online resources.

This lab is a bonus lab, and is only for students who have completed ALL of labs 1 to 24 by February 14th

Lab Work

You may choose one of the following tasks, or discuss any option of your own choosing with a demonstrator:

1. Take the code for Lab 19 (calculator) and improve it so that the calculator can correctly process the following input sequences :

```
6 =
8 + 2 + 4 =
3 + 10 = - 4 =
-6 + 9 =
3 + 4 = + 7 =
```

2. Draw the Triangles from Lab 8, including the grid.

3. Improve Lab 9 with a Graphical User Interface, loops and file storage.

4. Implement an insertion or selection sorting algorithm on an array or an ArrayList.

5. Any other task which gains demonstrator approval e.g. choose a task from Project Euler https://projecteuler.net/

Lab 25 Completed

The code I present for my Lab 25 mark is my own work according to the definition on page 7 _____

☐ comments ☐ working ☐ submitted

Date **Demonstrator's Initials**

You made it - well done! We hope you feel that you have learned a lot about Java, and about programming, without being driven entirely mad in the process. Good luck with the final exam.

Executable Java programs

Any of your Java programs can be used to create a Java archive (jar) file. No matter how many classes are involved in the program, it will make just one jar file. A jar file is executable so is a convenient way for an end-user to use a Java program.

A jar file is transportable to other operating systems, so can run on Microsoft windows as well as other Macs. The operating system of any host machine will need to have Java installed, which keeps iPads out of the picture. Android devices are Java-friendly though.

Making a JAR file in DrJava

1. **Make sure you have compiled your code, so the .class files exist.**
2. **In DrJava, Project > New**

   Save As : – Choose a name and navigate to where you want your package saved. **Save.**

   In the **Project Properties** window which appears:

   a. **Project Root** – Defaults to Project's directory - Navigate to the directory where your `.class` files are saved. If you are working in a package, navigate to the directory **containing** the package directory.

   b. **Working Directory** – Defaults to Project's directory - Navigate to where your `.class` files are saved. If you are working in a package, navigate to the directory **containing** the package directory. Make sure the final directory name isn't doubled when it shows in the field – fix manually if necessary.

   c. **Main Class** – navigate to the application class

   If you are working in a package, it will show as `packageName.applicationClassName`
4. **OK.** Your project is made, and can be tested at this point by compiling and running if you wish.
5. **In DrJava, Project > CreateJarFileFromProject**

   Check ف Jar classes

   Check ف Make executable

   Main class – should be showing

   If you are working in a package, it will show as `packageName.applicationClassName`
6. **Jar File** – this is the classpath and the filename for your executable file. **Navigate** to where you want the file, and give it a name. **Save. OK.** Your jar file should be ready for use.

Making a JAR file in Terminal

For this example, we will assume your code is in a package, the package is called `brick` and your application class is `BrickApp`. So your `.java` and `.class` files are in a directory called `brick`.

1. Make sure you have **compiled** your code, so the `.class` files exist.
2. **In any application, make a file called `Manifest.txt` which contains the text `Main-Class:` followed by the name of the application class AND a line return at the end of the instruction.**

   e.g. `Main-Class: brick.BrickApp`
3. In order that your setup matches the instruction in Step 5, **save** the `Manifest` file in the same directory as your `.class` files (e.g. inside the directory called `brick`)
4. In `Terminal`, **navigate** to the directory **containing** the package directory.
5. **Type the instruction** `jar cfm outputFilename.jar brick/Manifest.txt brick/*.class`

   Your jar file should be ready for use.

A brief explanation of the `jar` instruction:	
`cfm`	**c**reate a jar file, output goes to a **fi**le, **m**erges info from existing files
`outputFilename.jar`	you may name the output file anything you please, and use directory paths to place it
`brick/Manifest.txt`	where to find the manifest file, and what its exact name is
`brick/*.class`	the class files for the program – can also be listed individually, separated by a comma

Readings

The following Readings cover material that is generally useful, or not described in depth in L,D&C (the text book). The first three (under the heading "Getting Started") focus on introductory topics that it may be useful to read at the start of the course. Readings marked below with a "*" are for information only (they aren't "examinable").

Index

Where do You Begin?

Learning to program can seem hard. Initially it seems like a strange way of thinking. OK – it is a strange way of thinking! But don't let that initial strangeness put you off. You should become comfortable with the ideas and skills of programming fairly quickly, and when you do you will find them very useful and powerful.

Programming is really about problem solving. A programming language, like Java, gives you a set of tools and ways of representing and thinking about problems and their solutions. The good news is that once you have learned your first programming language the worst is over, it's much easier to learn subsequent languages.

Java is an Object–Oriented (OO) programming language. You write the code for the classes in your program. Programs work by creating "instances" of classes which are called objects, and then using these objects to get things done. (So one way of thinking about classes is as "templates" or descriptions of objects that can be created). Your program can create objects which are instances of classes you have written, or instances of pre–existing classes in the Java libraries. To learn Java we need to learn about this OO way of thinking, problem solving and programming.

For most of you, this is your first programming language. The hardest thing about writing your first few programs is getting started – looking at that blank sheet of paper and thinking "now what?". Where do you begin? In COMP160 we try to help by describing labs as a clear sequence of steps to complete. You should also read, and use, the COMP160 Program Development Process described in the Reading "How to Write a Program", page 119.

A good way to begin is by writing down the problem in your own words. Start by paraphrasing the problem and then work on breaking down the task into descriptions of smaller issues. (This method of breaking big problems down into successively smaller problems is often called a top–down approach to problem solving). Doing this will help you see patterns and similarities as well as pinpoint areas that will be more difficult.

You don't need to write all the code of the program in final and perfect form right away. You can develop it from an initial outline, filling in details later on. Steps such as specifying the algorithm, or sketching out parts of the program in rough code (or "pseudocode") are a very good place to start.

L,D&C can't teach everything at once! They could have spent longer at the start on describing what objects are, and how to use them to write programs. So the next Reading is a brief introduction that might help to get you started thinking about OO programming.

Finally, there are lots of ways of solving problems and writing programs. The OO approach is only one of these. It may be that other more general problem solving methods (that are not specifically related to objects) will be useful sometimes or help get you started. There is a Reading ("Problem Solving") below on general problem solving methods.

Last thing. You write programs to do some task or solve some problem. Follow the COMP160 Program Development Process! The first step is always always always to start thinking about the problem first and how to solve it. Think first, write code later. Keep in mind the First Rule of Programming:

> The sooner you start coding the longer the program will take to write.

Seriously! If you think first and develop a good understanding of the problem and a good design for the program then the hard work is done. The code should be easy to write and you should create a clean, well organised program.

Object-Oriented Design

Java is an Object–Oriented (OO) programming language. Writing a Java program involves writing the code for the classes in the program. A running program is a collection of interacting objects which are built from the classes. In Java there are two kinds of objects:

> Class objects: One class object is automatically created for each class.
>
> Instance objects: Any number of instance objects ("instances") may be created for a class. So classes can be thought of as "templates" or descriptions of objects that can be created. Instance objects are the usual objects that people mean when they talk about OO programming.

For the early programs in COMP160 we keep things simple by writing just one class, so our programs consist of the one corresponding class object (and various class and instance objects automatically created from library classes). We don't actually write classes for our own instance objects until later (Lab 6, Lecture 6).

However, OO design issues and discussions are almost always about instance objects, so we need to be thinking about them right from the start. Make sure you read the L,D&C Section 1.5 "Object–Oriented Programming" for an introduction and a hint of topics to come.

What is an (instance) object?

In practical terms an object is an instance of a class that gets created (by the Java interpreter) when your program is run, and used in the ways determined by your program. But what is an object in terms of problem solving? What is the best way to think about objects? There are many definitions, and no one best or correct way. Here is one definition:

> "We define an object as a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand." (J. Rumbaugh *et al.*, Object-Oriented Modeling and Design, Prentice Hall. 1991).

An object is a useful chunk of the problem or its solution.

Many text books suggest that it is useful to think of objects as agents – independent entities that can store information and do various things. They usually present this idea with some kind of analogy, like this one (adapted from T. Budd, Understanding Object-Oriented Programming With Java (updated edition), Addison-Wesley. 2000):

> Suppose I want to send flowers to my friend Sally in Auckland. I can't drop them off myself. But I can stop by and visit Fred the local florist. I tell Fred the kind of flowers I want to send, pay for them, and Fred arranges for them to be delivered to Sally in Auckland.
>
> To solve my flower problem all I had to do was find the right agent, Fred the florist, and give him a message containing my request. It's Fred's job to carry out my request. He has some method of doing this (some process or algorithm that he follows). I don't know, and I don't need to know the details of Fred's method (so long as it works and Sally gets her flowers). Fred's method for carrying out my request is hidden from me.
>
> What Fred really did was probably something like this. Fred probably called up a florist in Auckland and gave her a message about my request. (That florist got flowers for her shop from a wholesaler, who in turn got them from several gardeners). She probably had an assistant make up the bunch of flowers that I asked for and call a delivery person. The delivery person took the flowers to Sally. So, unbeknown to me my request was carried out by a whole community of interacting agents (florists, assistants, gardeners, wholesalers, delivery people). Each of these agents can remember certain things, and can do certain things, and can send messages to other agents.

So how do you think of an OO program? Budd says it very well:

> An object-oriented program is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

Object-oriented concepts

That was all fairly general and introductory. The notes in this section are more detailed, and they won't necessarily make sense early on in the course. But this is a section that it will be worthwhile re-reading from time to time. As you get into OO programming the advice and principles described here should start to make sense, and become very useful.

Behaviour and State: The behaviour of an object is the set of actions it can perform, in effect its methods. The state of an object is all the information held within it, in effect the values stored in its data fields. States can change over time. Almost everything that we need to know about an object is contained in its state and behaviour!

Cohesion and Coupling: Cohesion is the degree to which the responsibilities of a single object form a meaningful unit. The tasks that an object performs should all be related in some way. Probably the most frequent way in which tasks are related is by the shared need to access the same data. Bundle data and the tasks that relate to it together into an object. This cohesion is a Good Thing. Coupling describes the complexity of the relationship between objects. In particular coupling is increased when one object must access the state (the data fields) of another object. Coupling is a Bad Thing – to be avoided. Try to move the responsibility for working with a certain piece of data in to the object that actually holds the data.

Interface and Implementation (Parnas's Principles): The sum of the ways a software component (such as an object) behaves is also called its interface. (A more technical definition is the set of exported names and signatures). Describing a component according to its interface has an important benefit – it is possible for one programmer to use a component developed by another programmer without needing to know the details of how it works (how it is implemented). The hiding of implementation details behind the interface is called information hiding, and it is a Good Thing. We say that the component encapsulates the behaviour.

This makes most sense when there are multiple programmers working on the same program. Each programmer has different views of the software components. For the components they develop they have an "implementation view" – they need to see and know all the details. For the components developed by others they have an "interface view" – they only need to know how to use those components (not how they work internally). By interacting with other programmers' components only via their interfaces then all the components should interact in planned, predictable, correct ways. This separation of interface and implementation is often described as the most important concept in software engineering!

This can all be summarised in Parnas's principles:

> The developer of a software component must provide other users with all the information that they need to use the component and no other information.
>
> The developer of a software component must be given all the information necessary to carry out the responsibilities of the component and no other information.

Remember – this does not all need to make sense early on in the course. Re-read this section every now and then, and bits of it will start to make sense. Hopefully by the end it should be clear, and it should help you to think about good ways of designing and structuring your programs.

General Problem Solving

You solve problems every day; you make decisions, prioritise, juggle and plan. A good problem–solver will likely make a good programmer. To solve a problem you need to have a clear goal in mind. To get to the goal, design a sequence of steps / actions (something like for example a recipe, or the instructions for setting up a video or a mobile phone). A formal description of steps is called an **algorithm**. Noted below are some commonly applied strategies from problem solving (adapted from Dale *et al.*, Programming and Problem Solving with Ada, DC Heath. 1994) that may be helpful...

Ask Questions

"I kept six honest serving men, They taught me all I knew, Their names were What and Why and When, And How and Where and Who." – R. Kipling.

Look for familiar things

You are good at recognising patterns and similar situations in everyday life. Try and apply the same skills to problem solving! Effective programmers will identify the elements of a task that must be completed to reach the final goal. They will recognise similar situations and patterns and re-use or modify existing solutions to a problem. So, if you needed to find from a list the highest and lowest mountain peak in the world you could re–use parts of a program that had been written previously to solve the problem of finding the highest and lowest temperature on record.

Solve by analogy

Spend time thinking about the problem at hand before you start typing at the keyboard, it will pay off in the long run. Broaden the strategy of looking for features that are familiar to look for analogies – general concepts that may transfer from other domains. For example, you might think of a warehouse inventory system as being like a library catalogue. The people who designed the first Mac user interface used the analogy of a desktop, and this general concept helped them to visualise, organise and implement the interface.

Means–ends analysis

Often the beginning and end state to a problem are given, and your task is to define how to get from one to the other. Suppose you want to get from Bluff to Kaitaia, that is you know where to begin your journey and where it ends, and you are now faced with the choice of how to get there. The means you chose will depend on your circumstances which is probably a trade–off between cost, time and other factors (how much money, how much time and so on). In general terms a means–ends analysis focuses the choice of intermediate steps on those that are well suited to moving from the start state to the end.

Divide and conquer

Break the large task down into sensible, manageable, achievable pieces. A good strategy to make seemingly overwhelming odds approachable; just one step at a time. Taken as a whole, the task of "preparing for exams" might seem impossibly huge. But it can be broken down into smaller tasks, preparing for each paper, revising each lecture, and so on.

Building-blocks

A combination of look for familiar things and divide and conquer. Divide the large task into smaller units and look for existing solutions that are applicable to these units. It maybe possible to put existing solutions together in some way to solve your current task.

Merging solutions

Sometimes we can save time by merging two tasks (or parts of them) in to one. If I need to bake a cake, and make scones, I could do one (get out ingredients, bake, clean up) and then the other (get out ingredients, bake, clean up). But parts of these tasks can be shared, and it is much more efficient to combine the two processes.

How to Write a Program

There is a lot of research specifically about how people learn to program. There are at least three different kinds of attributes that you need: **Knowledge** of the programming language (e.g. knowing what a "for" loop is); **Strategies** for using the knowledge (e.g. knowing when it is appropriate to use a "for" loop in designing a program); and **Models** of the program (a clear vision of the program that you are trying to write, and if it doesn't work, an understanding of the program that you have actually written instead!).

COMP160 Program Development Process

In practical terms, we recommend the COMP160 Program Development Process described here:

1 **Establish the requirements (understand the task):** We try to explain tasks clearly in the lab book, but it is still vital to understand the task - you can't write a program if you don't know what it's for!

2 **Design the program:** Identify the objects that are required (that naturally match parts of the problem). What data does each object / class need (data fields)? What things does the object /class need to be able to do (methods)? Can you use any existing classes from the libraries (instead of writing everything yourself)?

3 **Implement the design (write the program code):** Write the code in parts / sections, running the code (testing its behaviour) frequently as you go. This makes it much easier to find and fix bugs.

4 **Test the program:** When you think the program is finished, test the whole thing. Test the range of inputs it is supposed to deal with (and also possible "bad inputs"!).

Years of experience show us that following this process results in better progress in labs…

Develop incrementally

Don't try and write a big program all in one go, then expect to compile it and run. There will be errors, and it's hard to find and fix all the errors in a large complicated program! Instead, build the program up incrementally. Write part of the program and test it thoroughly. Write a simple version of a method or class, and when it is working add further detail. In short – try to always build on a firm foundation!

Write safe programs

This topic is so important that it gets its own Reading, page 123.

Think about constraints

One tool for helping with program design is to think about constraints, such as **preconditions** and **postconditions**. Preconditions are things that must be true before a method (or any other logical grouping of code, but here we will use methods as an example) is run, and postconditions are things that must be true after a method has run. A related concept is **invariants**, things that must remain true while a method executes.

It is useful to think about and note preconditions and postconditions when designing methods. For example, in some imaginary toy banking program (where accounts have no overdraft, so must have a balance of $0 or greater), a method for withdrawing an amount from an account might look like this:

```
/* precondition: data field currentBalance must contain a value which is larger
   than or equal to the parameter withdrawAmount
   postcondition: data field currentBalance is set to the old balance, less the
   withdraw amount, this value is not negative
*/
public void withdraw (double withdrawAmount) {
   currentBalance -= withdrawAmount;
}
```

In some languages preconditions and post conditions can be expressed in the code as actual parts of the program. In Java they are usually just constraints to be aware of, and possibly include in comments or other descriptions of the code. Recent versions of Java also provide a mechanism called **assertions** which allow you to code pre and post conditions during program development (they require the Java compiler to be called in a certain way) – see for example: http://www.deitel.com/articles/java_tutorials/20060106/Assertions.html

Look for patterns

You don't have to reinvent every wheel. Lots of programmers have solved lots of problems, and you can use their work! This includes code (see below), but also useful patterns (designs). Useful patterns exist at the level of parts of programs, and at the level of the overall designs of programs.

For example, loops can be described in terms of their mechanics ("for", "do", "while" and so on). Or they can be described in terms of their design, such as counter controlled, state controlled, or sentinel controlled. A counter controlled loop repeats a certain number of times depending on the value of a counter variable. You can use any loop in the language as a counter controlled loop (but "for" loops are particularly convenient). An abstract pattern for a counter controlled loop is:

```
initialise counter variable
test counter variable and repeat loop if necessary {
   loop body
   update counter variable
}
```

A state controlled loop repeats until a certain state is achieved (either true or false as desired). This state is usually represented as a boolean condition / test (e.g. `currentPosition < length`) or the value of a boolean variable (e.g. `success = true`, such a variable is often called a "flag" because it flags an important condition). An abstract pattern for a state controlled loop is:

```
test condition or flag and repeat loop if necessary {
   loop body
   update the value of the test condition or flag if necessary
}
```

A sentinel controlled loop is usually used to read in data, particularly when we don't know in advance how much data to read. A sentinel is a special value that is used to signal the end of input (usually an "impossible" value for the input, e.g. -1 if the inputs are all positive). An abstract pattern for a sentinel controlled loop is:

```
read the first data value
if the data value is not the sentinel repeat the loop {
   loop body / process the data value
   read the next data value
}
```

There are patterns at many levels of program design, including the level of the overall design of programs. Many apparently different tasks have the same underlying structure, and there are well known, good designs for solutions. These design patterns can give a programmer excellent guidance on the structure of a program, and save a lot of time! Google for "design patterns", or see: http://www.javacamp.org/designPattern/ or examples here http://www.javaworld.com/columns/jw-java-design-patterns-index.shtml .

Look for code

There is lots of very useful code out there. The whole philosophy, and many of the practical advantages of the OO approach to programming, are about making code easier to develop and reuse.

A huge amount of useful code is already formalised and available as the Java libraries / APIs (see References at the end of this lab book). You can use the libraries for many common tasks (like searching or sorting) instead of writing your own code. There is also a lot of freely available code out there on the net (but this might be of variable quality, and of course it shouldn't be used for assignments, see the University policies on plagiarism). The Internet and WWW are largely built with free and open programs, and this philosophy of making code available is a strong tradition among programmers.

Debugging Code (and DrJava Tools)

One of the skills of programming is finding and fixing problems (debugging). Programs are complicated, and code that you wrote may not actually be working in the way that you expect! Debugging a program can be a bit like detective work, you may have to look for hints and clues, you may have to conduct a systematic study, you have got to work out what is really going on.

Error messages are your friends

If your program fails to compile, or runs and crashes, you will get an error message (these are called, respectively, compile time and run time messages). At first such messages may seem like gibberish, but they are really extremely useful, and as you gain experience you will find it easier and easier to interpret them. So, read the error messages! They tell you exactly what is wrong (if you can interpret it!).

If you need a bit of help in interpreting error messages, the Web is a great resource. I wish I had a dollar for every problem I have solved by Googling for the text of an error message (almost always you can find a useful discussion somewhere), or see for example resources like http://mindprod.com/jgloss/errormessages.html .

However, the best advice about errors is to protect yourself in advance – Write Safe Programs (see the Reading page 123).

println is your friend

If you have a program that runs but doesn't work as expected (or has a run time crash that you really can't fix using the error message), then you need to find out more about what is going on. The simplest place to start is by printing out relevant information. In Java terms this means adding a few println statements, so we can call this println debugging. (The language C prints with printf, and if you Google for "printf debugging" you will find lots of resources and debate!). You can use `System.out.println`, but Java also supplies `System.err.println` (and DrJava helpfully colours this output red!).

For example, say you have a program with unexpected behaviour. You suspect the problem may be somewhere in `MyMethod`. So add printlns to print out the inputs to `MyMethod` when it is called - are these what you expected? Add printlns to check the outputs - are these what you expected? (You are checking aspects of pre and post conditions, see the Reading "How to Write a Program" page 119). If something is wrong, investigate further. If `MyMethod` uses some variables, print them out at various points in the method. If it has a loop with a counter, maybe check the behaviour of the counter by printing it out at the start and end of the loop (or even every iteration). Often this kind of approach is all that is needed to track down a problem.

Similarly, if your program runs but crashes in `MyMethod`, add printlns at various points ("I got to here 1", "I got to here 2" and so on). Work out which line is causing the crash. What variables or operation is involved with the statement on that line? Before it gets executed, print out relevant details so that you can check them.

Use other tools

There is plenty of debate about debugging. Simple println / printf debugging gets much less useful as programs get large and complicated (and some people think it should never be used at all!). But there are also plenty of other tools you can use. Lots of languages have support programs that help to write and debug code (such as the popular "Lint" program that helps to pick the fluff out of C). In most cases an IDE will provide tools to help develop and debug programs, and here we will very briefly consider some aspects of our DrJava (see also the DrJava userdoc in LabFiles or download it from http://drjava.org/).

The DrJava interactions panel

The DrJava interactions panel is very useful. It lets you test bits of Java code without having to put them in a whole program. This is particularly useful for exploring the behaviour of objects, e.g. see Lab 7 where we use the interactions panel to make instances of various JFrame objects and explore their behaviour. You can use this to test the classes that you write! If you have written some support class, use interactions to make an instance, call the various methods with sample input to test them and make sure that they behave as expected. Then (with confidence that this class is safe) you can go on and write the rest of the program.

You can also use interactions to test most other aspects of Java code. Print out the result of a quick calculation to test the behaviour of a mathematical operator. Type in a "for" loop line by line to see how it will behave. All very useful! (DrJava is actually pretty cunning, it runs two copies of the Java virtual machine / interpreter, one to support most of the application, and one just for the interactions panel).

The DrJava debugger

DrJava supplies a built in debugger (similar examples are provided by many IDEs). This automates and greatly extends the kinds of exploration of a program that it is possible to do with println debugging. It makes it easy to examine the behaviour of a running program and the state of its data.

Common debugger tools are: stepping through code (executing it one line at a time), break points (executing code as normal but pausing at specific selected points), displaying the values of selected variables as the code executes, and even manually changing the values of variables as the code executes (to test the behaviour of the code as desired). This is much more powerful than println debugging, and really lets you see what is going on as the program runs! See the DrJava userdoc, Chapter 9, for more details, or just experiment with a simple program...

Other DrJava

DrJava, like many IDEs, supplies a facility called "projects" for managing very complicated programs with large numbers of files. We won't use it in COMP160.

Writing Safe Programs

A working program is not necessarily a good program. A good program should not only work, it should work safely – by which we mean that the program should be designed to handle possible problems without crashing. Programs crash when they execute commands that damage the operating system or the contents of memory, or try to execute commands that make no sense (like trying to divide a number by zero, or trying to read from a file that isn't there).

Java is a great language for writing safe programs because (1) it provides type checking and other kinds of checking that identify problems when the program is compiled, and (2) it simplifies and automates some of the most dangerous features of some programming languages (instead of pointers and manual memory allocation Java has references and automatic garbage collection), and (3) it provides language features that let the programmer test for and handle problems while the program is running, i.e. "exceptions" and the "try..catch" system (L,D&C Chapter 10, Lec 20, and below). We only cover only a bit of the try..catch system in COMP160 – if you are planning to go on with Java programming read L,D&C Chapter 10 some time. (Read the other chapters too!).

Handling problems

When a Java program encounters a known problem (such as trying to read an `int` value and finding that the input is not an `int`) it generates an "exception" (error), and constructs an exception object (e.g. an "InputMismatchException"), which is a special object describing the problem (much like the Event objects used in the event model). At this point, depending on how the programmer has written the program, one of three things will happen:

(1) If the programmer hasn't set up a `try..catch` for the exception the program will crash.

(2) If the problem occurs inside a `try` block the program will execute the code in the `catch` block (see examples below).

(3) If the method throws (propagates) the exception it can be caught elsewhere in the program (not done in COMP160).

In this Reading we will see some examples of the second case. The code in the `catch` block should handle the exception, possibly by printing a message to the user, and / or taking some action to recover from the problem, and / or stopping the program with `System.exit(1)` (exit with non 0 values indicates that the program is stopping because of a problem). The catch block could also use information in the exception object by calling various methods on it.

To write safe programs there are:

(A) various places where we should think about things that could go wrong

(B) various places that we should think about things that could go wrong and use try..catch

(C) various operations that are so potentially unsafe that the Java compiler forces us to use try..catch.

Let's look at examples:

Case A: Tests you should make

In most programs of reasonable size, there are many places where you can add extra safety checks. For example, if you're accessing an item at position `n` in an array of 100 elements, you could check to see (use an `if`) that `n >= 0` and `n <= 99`. If the user has entered a value `x` that should be in the range 1 to 5, you should check to see that `x >= 1` and `x <= 5`. If you are using a scanner to read `int` tokens from a string, you should check to see (use the `hasNextInt` method) that it has another `int` token before you read it. This is also a great way to process all the ints in a string without needing to know how many there are – the following loop will print (on new lines) every int in the string.

```
Scanner line = new Scanner("1 2 3");
while ( line.hasNextInt() ) {            // hasNextInt returns true or false, if true enter the loop
  System.out.println( line.nextInt() ); // nextInt returns the next int
}
```

In Lab 20 Part 2 there is a very nice example of a check to see that the structure of a string has the correct sequence of tokens before proceeding to process the string. There are many more cases like this where you can make safety checks in your programs!

Case B: Tests you should make with try..catch

Processing data that has been input to the program (e.g. from the user, or from a file) is a very common source of problems. Users and files are outside the program's control, so the program might not get the data it expects!

For example, the `readInt` method below is unsafe. If the user does as asked and inputs an `int` value it works fine, but if the user inputs something else (such as a character or a string) the program crashes with an "InputMismatchException".

```
public static int readInt() {
   Scanner sc = new Scanner(System.in);
   System.out.println("Please enter an integer);
   return sc.nextInt();
}
```

Below is a safer version of `readInt`. It follows a common general pattern for dealing with input safely:

(1) create a loop that will repeat until the input has been tested and found to be safe / correct

(2) within the loop read input and test to see if it is correct

(3) if it is correct exit the loop

(4) if it isn't correct repeat the loop and try to read more input

```
public static int readInt() {
     boolean success;          // indicates whether we have correctly read an int
     int input = 0;      // this is the variable that the method will set, and return the value of
     do {                // loop as often as necessary to successfully read int
        success = true; // assume that this attempt will be successful
        System.out.println("Please enter an integer");
        try {
           Scanner sc = new Scanner(System.in);       // make a scanner
           input = sc.nextInt();                 // try to read an int
        } catch (InputMismatchException e) {    // this block will be executed if read fails
           System.out.println("Unexpected input, please try again.");
           success = false;    // indicate failure so that the loop repeats
        }
     } while (!success);       // if success is false, repeat the loop
     return input;             // else exit the loop, input is correct so return it
  }
```

In this example the key testing of the validity of the input (is it an int?) is done by the statement

```
input = sc.nextInt();
```

If the input is invalid Java generates an "InputMismatchException" and executes the catch block (which here doesn't do anything with the exception object `e`, but does take other action).

Note that the variable `input` must be declared and initialised before the `try..catch` block.

Case C: Tests you must make with try..catch

Files are notoriously dangerous sources of input (they might have moved or changed since the program was written), so in general code that accesses a file must be in an appropriate try..catch, or the program will not compile.

The code on page 126 which deals with the file is in a try block. The first line attempts to create a scanner for the given file (identified by the string `fileName`, e.g. `data.txt`):

- if the scanner is successful (the file exists and can be opened) we carry on in this block
- otherwise a FileNotFound exception is generated and we execute the catch block, which prints a message to the user and stops the program.

Note: When working with files see the Reading **"Locating Support Files"** page 128

Java Input and Output (I/O)

Java is descended from a language that was designed for embedded systems (like cell phones and dishwashers). It therefore provides very complex and fine control of input and output (I/O) from various sources (the user, files, various devices, even the Internet). In older versions of Java this flexibility led to complexity in tasks that should be simple, like reading input from the keyboard or writing output to the screen. Fortunately Java 5.0 (i.e. JDK 1.5) introduced the useful new Scanner class, which provides a much more convenient way of doing these tasks.

Streams

Before we get into the details of Scanner, it is useful to have a bit of background on the concept of streams. In computing, "stream" is used in various ways, all referring to a succession of data elements over time (e.g. "streaming media"). Java follows the style of the Unix (and Linux) operating systems, which use the concept of a stream (source or destination of a sequence of data) as an abstraction for dealing uniformly with files and devices. Java (in the Unix tradition) has three "standard streams":

in for reading data from the keyboard, e.g. System.in

out for writing data to the console, e.g. System.out.println

err for writing error messages, e.g. System.err.println (also to the console, in DrJava the output of System.err.println() is coloured red!).

Other streams can be defined as needed, to interact with devices, memory, files, strings, and so on. This used to be complicated, but the Scanner class, simplifies many aspects of reading streams (hides the details).

Reading input from the keyboard

Any class that uses Scanner must **`import`** `java.util.Scanner;` (include this statement at the start of the file containing the class). To read from the keyboard (actually via the console, as described in the next section) make a Scanner object as shown below, and call a method on it to read the data type required. This example shows reading an integer (int) value.

```
Scanner in = new Scanner(System.in);
System.out.println("Please enter an int: ");
int i = in.nextInt();
```

To read different data types just call different methods on your scanner object, e.g. to read a double call `nextDouble` (see the other scanner methods listed in L,D&C Section 2.6 Fig. 2.7 page 62).

Reading an input like this is simple, but it means that the programmer must always remember to print a prompt to the user, to tell them what kind of data to enter.

Writing output to the screen / console

Java, like most programming languages, can be run in a command line window / terminal which is supplied by the operating system (you do this, for example, in Lab 15). In this case all text input and output happens in the terminal. When Java is run via a graphical user interface or IDE (Integrated Development Environment) like DrJava, then that

interface or IDE must supply a display for basic text input and output. This display is called the console. DrJava provides a console in a complicated way. Outputs are shown in the interactions tab, inputs are read using a popup window. Messy!

Writing text output to the console is simple, use the method System.out.println. For example:

```
System.out.println("Hello world.");
```

The println method can accept any number of inputs of different data types, which get joined together (concatenated) into an output string, which is then displayed in the console.

Files

When working with support files (data, image or other files used by your program) see the Reading below on "Locating Support Files" page 128. Here we assume that the support files and your program are in the same directory, and that Java is correctly configured to use that directory. Reading and writing to a file are briefly covered in Lecture 20.

We will work only with "text files", where the contents are treated as a sequence of characters. Connected sequences of visible characters are called "tokens", tokens are separated by "white space" (invisible characters like blanks, tabs or new lines). For example, the following illustrations of files all contain three tokens:

```
token1  a  42
```

```
long-example!
42
@@@@
```

```
22      23
24
```

Reading input from a file

Any class that works with files as shown here should **`import`** `java.util.Scanner;` and
`import` `java.io.*;` (include these statements at the start of the file containing the class).
In this example we assume a support file called `data.txt`. If we want to read an an integer from the file we can do it as follows:

```
String fileName =  "data.txt";
int nextItem = 0;
try{
  Scanner fileScan = new Scanner(new File( fileName)); // make a scanner for the file
  while (fileScan.hasNextInt()){
    nextItem = fileScan.nextInt(); // read the next token as an int
  }//while
} catch (FileNotFoundException e){
  System.out.println("File not found. Check file name and location.");
  System.exit(1);// stops the program - can't proceed without a file!
}//catch
```

The main steps are to create a scanner for the file, and read the next token in the file as an int. These steps must be contained in a try..catch block – see the Reading "Writing Safe Programs" page 123. Using for example the third of the example files described just above, this code would read each token as an `int`, and write out the values 22, 23 and 24.

The while loop checks that there is another int to read before going ahead and reading it. To read different data types call different methods on your Scanner object, e.g. to read a double call `nextDouble` (see the other Scanner methods listed in L,D&C p. 86). A common technique is to use `nextLine` to read a whole line of the file as a string and then work within the program to get the tokens of the string – see the section "Working with tokens" page 127 (and see for example Lab 20).

If the next token in the file does not match what you are trying to read (e.g. you call `nextInt` when the next token is "Abc") then the read will fail with an "InputMismatchException". You should do the extra work to write programs that are safe from this kind of error, see the Reading "Writing Safe Programs" page 123.

Writing output to a file (and other operations)

We don't write to files in COMP160, but we should note it in passing. Writing output to a file is a bit more complicated than reading (the language does not yet provided a simple tool like Scanner for writing to files).

The example in L,D&C Listing 10.7 page 457-8 shows the basics – construct a `FileWriter` that uses a `PrintWriter`, call `print` and `println` methods to write to the file, and always close the file when you are finished (if you don't, you can in some circumstances damage your computer's file system, or more likely you will find that not all of the data "written" to the file is there).

Note that this example does not `try..catch` any possible exceptions, it just "throws" them (see the declaration of the main method), i.e. passes them off for some other part of the program to deal with (see L,D&C Ch 10).

There are many other operations you can do with files. Check if they exist, create them, create new directories, rename them, check to see if they are readable or writable, and so on. For a brief summary see the File class (package java.io) in the Java Libraries / APIs.

Working with tokens

We often want to process the tokens in a string of text (e.g. a string read in from the user or from a file). The most convenient method is to create a Scanner for the string (remember to **`import`** `java.util.Scanner;`). In this example we just use an example string containing a first name, a last name, and an imaginary age:

```
String line = "James Dean 21"; // example input string
Scanner sc = new Scanner(line); // create scanner to process input string
String first = sc.next(); // call methods to return tokens from the string
String last = sc.next();
int age = sc.nextInt();
System.out.println( age + ": " + last + ", " + first);
```

The example shows how to read the first token as a string, the second token as a string, and the last token as an int. The final statement would write the output "`21: Dean, James`". The process does not alter the original string, the value of `line` remains as set by the first statement. See also the Reading "Writing Safe Programs" page 123.

There are many other things that can be done to process tokens. See for example L,D&C 2.6, 4.6, and for a brief summary see the Scanner class (package java.util) in the Java Libraries / APIs.

Locating Support Files

Java is supposed to be "cross platform", i.e. run on different kinds of operating system, computer or device. One of the most problematic aspects of making this work is interacting with files, because different operating systems have different ways of managing files.

CLASSPATH

To support Java an operating system needs to set a CLASSPATH variable specifying the directories where Java class files can be found. Java is supposed to handle CLASSPATH issues automatically, but problems sometimes arise (in fact they account for many of the problems using Java on different machines). CLASSPATH always includes the Java libraries, and the directories where any source code (".java" files) get compiled. You can find out more about CLASSPATH here:

`https://docs.oracle.com/javase/tutorial/essential/environment/paths.html`

Support files

It is very convenient to put a program (source code) and any other / support files (like images, or data files) that it uses together in one directory. Then we can access a support file from our Java program using its simple name e.g. `"data.txt"` or `"arrowUp.gif"`. L,D&C do this and we will do this in COMP160. However, this is not guaranteed to work, getting a program to locate other files can be another source of complication in Java.

The complication arises from the concept of the "current directory", which is where Java looks for files using a simple name. For different Java compilers, environments or IDEs the current directory may be the directory where Java was launched, where the IDE is installed, or where your source code is (the only case where simple names will work). You can find out what Java thinks the current directory is by using the System property `"user.dir"`. e.g.: `System.out.println(System.getProperty("user.dir"));`

DrJava is is pretty good at setting the current directory in a useful way, and should set it to the directory where your source code is, and therefore simple names should work. (If not, try launching DrJava from that directory, e.g. from the command line / terminal, or move the application there and launch / double click.)

L,D&C has been written for the case where simple names work. So for example the DirectionPanel example in L,D&L Chapter 6 (Listing 6.28 page 307) loads an image file by referring to it with a simple name `"arrowUp.gif"`. But for some IDEs / cases with a different current directory setting this case will fail. In short, the L,D&C approach usually works, and it is nice and simple for teaching purposes, but it is not safe, so is teaching bad habits.

Safer approaches

A better approach would be to write a safe program, one that does not assume that the Java current directory happens to be set correctly. One clumsy way is to specify files not with their simple name, like `"data.txt"`(which is always relative to the current directory), but with their full path name, like (for example)
`"C:\My Documents\MyJavaWork\data.txt"` (which exactly locates the file). This works, but it means the code has to be rewritten every time a directory gets moved - not so good.

A better option is to use Java tools to work out the full path names for us! The `MyLoader` class shows how to use Java tools to turn a simple name into a full path name for any file anywhere in the CLASSPATH. Because CLASSPATH always includes the directory with the source code, we can carry on with the convenient practice of putting source code and support files in the same directory.

The `getPath` method takes as input a string representing a simple name, and returns a string representing a full path name. Thus anywhere that we can use a string to specify a file name, we can instead use the output of the `getPath` method.

For example, to create a scanner for a data file, where the usual version is:

```
Scanner in = new Scanner(new File("data.txt") );
```

our safe alternative is:

```
Scanner in = new Scanner(new File(new MyLoader().getPath("data.txt")));
```

Code for the `MyLoader` class is below. It could be used as an inner class. It extends the abstract interface `ClassLoader` from the package `java.net.URL`

```
/* Anthony Robins, May 2018, JDK1.8
   This class is used to help locate support files.  It has one method, which takes as input
   a string that is a simple name of a file, and returns a string that is the full path name of the file.
*/

import java.net.URL;

class MyLoader extends ClassLoader {  // needs to be static if used from a static context like main

   public String getPath(String fileName) {  // takes as input a string which is a simple file name
      URL url = getClass().getClassLoader().getResource(fileName); // get url for the file
      if(url == null) {    // handle the case of file not found
         System.out.println("File not found.");
         System.exit(1);   // stops program
      }
      return url.getPath(); // return a string which is a section of the url that is the full path name of the
                                                                                                    file
   }

}  // MyLoader
```

The ArrayList class

The `ArrayList` class is part of the Collections API, a group of classes that serve to organise and manage other objects. The `ArrayList` class is part of the `java.util` package of the standard class library. It provides a service similar to an array in that it can store a list of values and reference them by an index. However, whereas an array remains a fixed size throughout its existence, an `ArrayList` object dynamically grows and shrinks as needed. A data element can be inserted into or removed from any location (index) of an `ArrayList` object.

Unless we specify otherwise, an `ArrayList` is not declared to store a particular type. That is, an `ArrayList` object stores a list of references to the `Object` class, which means that any type of object can be added to an `ArrayList`. Because an `ArrayList` stores references, a primitive value must be stored in an appropriate wrapper class in order to be stored in an `ArrayList`. Autoboxing – the automatic conversion of a primitive type to its corresponding wrapper object – will occur when you assign a primitive type to an `ArrayList`, for example an `int` will automatically become an `Integer` object.

Some methods of the `ArrayList` class of the `java.util` package:

`ArrayList()`
Constructor: creates an initially empty list

`void add (Object obj)`
Inserts the specified object to the end of this list

`void add(int index, Object object)`
Inserts the specified element at the specified position in this list

`void clear()`
removes all of the elements from this list

`Object remove (int index)`
Removes the element at the specified position in this list

`Object get (int index)`
Returns the element at the specified position in this list

`int indexOf(Object obj)`
Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element

`boolean contains(Object obj)`
Returns true if this list contains the specified element

`boolean isEmpty`
Returns true if this list contains no elements

`int size()`
Returns the number of elements in this list

The program listing below instantiates an `ArrayList` called `course`. The method `add` is used to add several `String` objects to the `ArrayList` in a specific order. One specified string is deleted, and another is inserted at the same index. As with any object, the `toString` method of the `ArrayList` class is automatically called whenever it is sent to the `println` method i.e. `course.get(1)` is the same as `course.get(1).toString()`.

When an element from an `ArrayList` is deleted, the list of elements "collapses" so that the indexes are kept continuous for the remaining elements. Likewise, when an element is inserted at a particular point, the indexes of the other elements are adjusted accordingly.

```
/*
 * Papers.java
 *
 * Demonstrates the use of an ArrayList object.
 */

import java.util.ArrayList;

public class Papers {

   public static void main (String[] args) {

      ArrayList course = new ArrayList();

      course.add ("COMP160");
      course.add ("ENG127");
      course.add ("BSNS106");
      course.add ("HIST102");
      course.add ("MATH160");

      System.out.println (course);

      System.out.println ("At index 1: " + course.get(1));

      int location = course.indexOf ("HIST102");
      course.remove (location);
      System.out.println (course);

      course.add (location, "HIST104");
      System.out.println (course);

      System.out.println ("Number of papers : " + course.size());
   }
}
```

OUTPUT

```
[COMP160, ENG127, BSNS106, HIST102, MATH160]
At index 1: ENG127
[COMP160, ENG127, BSNS106, MATH160]
[COMP160, ENG127, BSNS106, HIST104, MATH160]
Number of papers : 5
```

Specifying an ArrayList Element type

By default, an `ArrayList` can store any type of object, so in order to retrieve a specific object from the `ArrayList`, the returned object must be cast to its original class.

For an ArrayList called `buttonList` which contains `JButton` objects:

```
ArrayList buttonList = new ArrayList();
buttonList.add(new JButton("Red")); etc.
```

the line `buttonList.get(0).getText()` will fail because the `Object` class does not have a `getText` method. When the object returned is cast into a `JButton`, there is no problem calling the `getText` method on it.

```
//these lines fail
 String text =  buttonList.get(0).getText() ;
 System.out.println(text);

//these lines work
 JButton red = (JButton) buttonList.get(0);
 System.out.println( red.getText());
```

An `ArrayList` object can be defined to accept only a particular type of object. The following line of code creates an `ArrayList` object called `transport` that stores `Vehicle` objects.

```
ArrayList<Vehicle> transport = new ArrayList<Vehicle>();
```

The type of the `transport` object is `ArrayList<Vehicle>`. Given this declaration, the compiler will not allow an object to be added to `transport` unless it is a `Vehicle` object (or one of its descendents through inheritance). In the `Papers` program we could have specified that the course was of type `ArrayList<String>`.

Declaring the element type of an `ArrayList` is usually a good idea because it adds a level of type-checking that we otherwise wouldn't have. In fact, the DrJava compiler produces a warning if the `ArrayList` is not type-specified:

```
warning: [unchecked] unchecked call to add(E) as a member of the raw
type java.util.ArrayList
```

Declaring the element type of an `ArrayList` also eliminates the need to cast an object into its true type when it is extracted from the `ArrayList`.

Sorting

The Collections framework has a useful `sort` method. To the `Papers.java` code, add the statements in bold type to sort the `ArrayList` in to lexicographic (alphabetic) order:

```
import java.util.ArrayList;
import java.util.Collections;

public class Papers {

   public static void main (String[] args) {

      ArrayList<String> course = new ArrayList<String>(); // type not required but desirable
            .
            .
            .
      System.out.println (course);

      System.out.println ("Number of papers : " + course.size());
      Collections.sort(course);
      System.out.println ("Sorted " + course);
   }
}
```

Reference Types

This Reading follows on from Lecture 14. In the notes below we provide more detail about the behaviour of reference types.

Data fields and variables have / "contain" either primitive values (a specific number, truth value or character) or reference values (a reference to a specific object, shown as an arrow on the following pages). Primitive values are of primitive types (e.g. int, double, boolean or char). Reference values are of reference types (classes in the program that we can construct instance objects of, including classes written by the programmer and Library classes).

In other words, to say that a data field / variable is of a primitive type X means that it can have a value that is of the primitive type X. To say that a data field / variable is of a reference type Y means that it can have a value that is a reference to an object which is an instance of class Y.

In this reading we will look at the behaviour of primitive types, reference types, and Strings (a particular kind of reference type) when we pass them as arguments, copy them using assignment, and compare them.

— Primitive types —

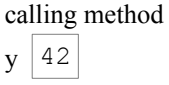
As a starting point for comparison, we will look at the behaviour of primitive types.

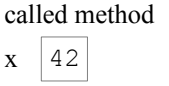
a) *passing arguments:* `x` gets a copy of the value of `y`

In this example we imagine that some method with a variable `y`, which has the value 42, calls another method called `aMethod`, passing it `y` as an input. As a result, the local variable `x` in `aMethod` gets a copy of this value. As x and y are separate variables, changing one does not affect the other.

```
// calling a method
aMethod(y);
```

```
// the method called
public void aMethod(int x){ ... }
```

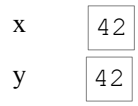




b) *assignment:* `x` gets a copy of the value of `y`

In this example we imagine that some method has two variables, `x` and `y`, and that `x` has the value 42. As a result of the assignment statement shown, the variable `y` gets a copy of this value. Again, x and y are separate variables, changing one does not affect the other.

```
x = y;
```



c) *comparison:* true if `x` and `y` have the same value

In this example we compare the values of the variables x and y. For the picture in case "b" just above, this comparison would be true.

```
x == y;
```

── Reference types ──

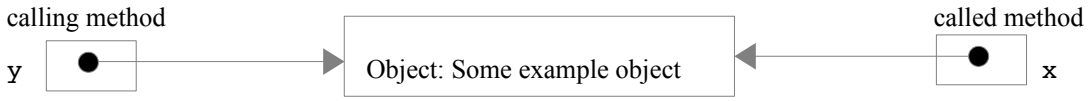
Variables of reference types (classes in the program) have reference values (a reference to a specific object).

An object is distinct and separate from a reference to the object.

For String, see the next page, otherwise...

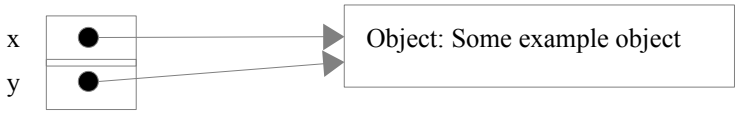
a) *passing arguments:* `x` gets a copy of the value of `y` (they refer to the same object)

```
//calling method                //the method called
aMethod(y);                     public void aMethod(Object x);
```



b) *assignment:* `x` gets a copy of the value of `y` (they refer to the same object)

```
x = y;
```



c) *comparison:* true if `x` and `y` have the same value (refer to the same object)

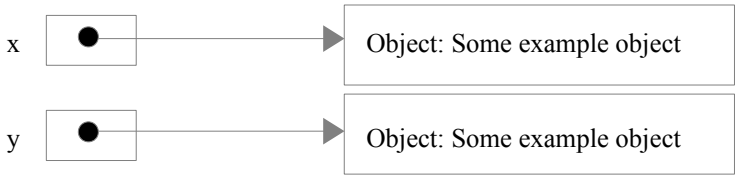
```
x == y;
```

in the pictures above `x == y` is true (in the picture below it is false)

Working with objects

If we want to copy objects themselves (rather than just the references to them) we can use for example the clone method (the example below makes object `x` a copy of object `y`). This can be very complicated.

```
Type x = (Type) y.clone();  // for some classes / types
```



We can compare objects using the equals method:

```
x.equals(y);
```

The default equals method tests for references to the same object (so it is the same as `x == y`). If you want to instead compare the states of objects to see if they are the same, then (1) for your own classes you must override this equals method (write your own!) to test for the states, but (2) most Library classes in (like String) have already done this and equals tests to see if objects are in the same state. (The states of Array objects can be compared with `java.util.Arrays.equals()`).

Most objects have default equals and clone methods inherited from `java.lang.Object`.

— Strings —

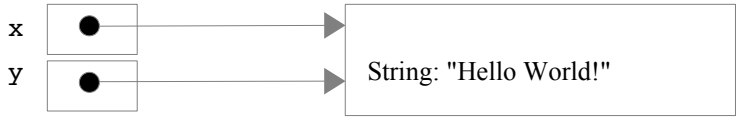
The class String includes several methods for working with string objects. Strings are immutable (see below):

a) *passing arguments*: `x` gets a copy of the value of `y` (they refer to the same string object)

```
//calling method          //the method called
aMethod(y);               public void aMethod(String x);
```

b) *assignment:* `x` gets a copy of the value of `y` (they refer to the same string object)

```
x = y;
```



c) *comparison:* true if `x` and `y` refer to the same string object

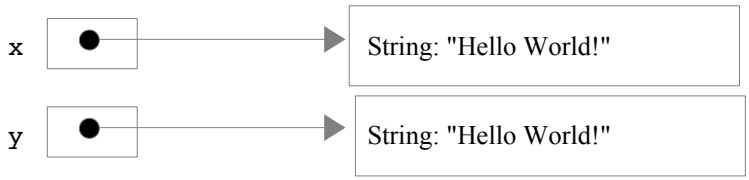
```
x == y;
```

in the pictures above `x == y` is true (in the one below it is false)

Working with String objects:

If we want to copy String objects themselves (rather than just the references to them) we can use for example:

```
String x = new String(y);
```

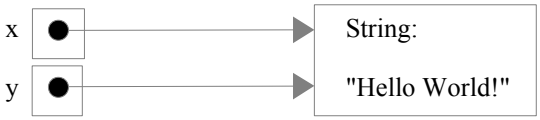


We can compare the states String objects using the equals method (the class String has overridden the default inherited from object):

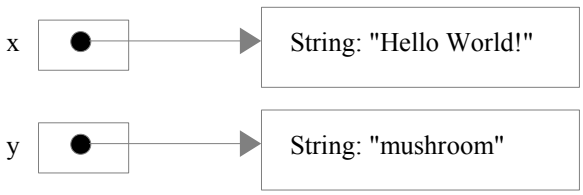
```
x.equals(y);      // is true in all pictures above because all the strings have the  same contents / state
```

NOTE: Strings and (and some other types defined by library classes) are immutable (never change their value). Instead of operations to change Strings, there are only operations to create new String objects. If `x` and `y` refer to the same String object and we assign a different value to `y`, then `x` and `y` now refer to different String objects (because `x` is not changed). For example:

On the left, after `x = y;`



On the right, after `y = "mushroom";`



Further note on the immutability of Strings.

In Java, Strings are handled in Pool format. For example:

```
String str1 = "xyz";
```

This string (`str1`) will be stored into memory in a particular address.

When you define a new String with same array of characters like this

```
String str2 = "xyz";
```

JVM will check in the String Pool whether a string containing those same characters is available. If there is a match the JVM will refer `str1` address to `str2`. Now the `str1` and `str2` refer to the same characters in the same memory location. This is a good for increasing memory efficiency. **But** - if you (could) change the `str1` characters, the changes would be reflected to `str2` because both `str1` and `str2` variables are referring to the same memory location.

To avoid this problem, Strings are immutable.

Style guide

These guidelines are part of a far more comprehensive document available at the URL:

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Line length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

When an expression will not fit on a single line, break after a comma or before an operator.

Each line should contain at most one statement.

Comments

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. Avoid duplicating information that is clear from the code.

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments should be used at the beginning of each file and before each method. All files should begin with a block comment that lists the class name, date, and author. Block comments inside a function or method should be indented to the same level as the code they describe.

```
/**
  This sort of comment should be used to head-up a method,
  explaining what it does, who wrote it, and when.
*/
```

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format.

```
// This sort of comment should be used inside methods for commenting "difficult" bits of code.
```

Very short comments can appear on the same line as the code they describe, but should be shifted far enough right to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same level.

Comments should be preceded by a blank line.

Variables

One declaration per line is recommended since it encourages commenting.

Initialize local variables where they're declared. The only reason not to initialize a variable where it is declared is if the initial value depends on some computation occurring first.

Put declarations only at the beginning of blocks (any code surrounded by braces).

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block.

Braces

The open brace "{" appears at the end of the same line as the declaration statement.

The closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement, when the "}" should appear immediately after the "{"

Statements inside braces should be indented at least one level more than the declaration statement.

Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This allows code to be easily added later. For example:

```
if (itemsRead == 0) {
  System.out.println("Nothing was read");
}
```

Spaces

One blank line should always be used between methods, between the local variables in a method and its first statement and between logical sections inside a method to improve readability.

A keyword followed by a parenthesis should be separated by a space.

A blank space should appear after commas in argument lists.

All binary operators except . should be separated from their operands by spaces.

A blank space should not be used between a method name and its opening parenthesis.

Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

The expressions in a for statement should be separated by blank spaces.

Parentheses

Use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems.

10 Tips For Beginner Programmers

1. Balance the quotes

When you type parentheses, double quote and other balanced symbols, type them in pairs e.g. `System.out.println("");` `public class Tester{}` `public void makeDo{}` then fill in the code required in between.

2. Use meaningful variable names

1. Don't use the same variable name twice in a class, even if they are used in different methods. It will make debugging easier.
2. When you name a class, variable, method, or constant, use a name that is, and will remain, meaningful to those programmers who will read your code.

3. Define symbolic names for constant values

By defining symbolic names for constant values, the program will be easier to read and modify. Suppose later you decide to change a constant: rather than change the value everywhere in your program, you can just change one value on your symbolic name directly.
e.g. use `array.length` as a test for when to end the loop rather than use the actual array size, e.g.10. If you need to change the array size later, you can simply change the size from the data field rather than change the size value everywhere within the program.

4. Initialise local variables

Initialise local variables when you declare them. e.g. `int a = 0; double b =0.0;` and `String today = "" ;`

5. Define small methods

Since there are few statements in a small method, it's easy to design, code, test, read, understand and use. Write a method to perform just one task rather than lots of tasks.

6. Leave a space between each section

Leave a space between class header and main method header, method header and method body and so on. It makes other people and you read the code more comfortably later.

7. Break up long lines

If you write a statement like the following, it's too long and complex:

```
System.out.println(adder(howLong("elephant","mouse"),7));
```

Write it in smaller steps like this:

```
int length = howLong("elephant","mouse");
double added = adder(value,7);
System.out.println(added);
```

8. Use `System.out.println()` help you to debug

When you are not getting the result you intended, `System.out.println()` can help you.
e.g.

```
public class TotalValue {
  public static void main(String[]args) {
    SomeClass s = new SomeClass();
    int total = sum(s.getFirst(), s.getSecond());
  }
  public static int sum(int first, int second) {
    //Use System.out.println() to check the parameter values.
    System.out.println("first is" + first);
    System.out.println("second is" + second);
    return first + second;
  }
}
```

9. Make sure your current code works before writing more

Before you do further programming, check your current code. Does it work? e.g. Suppose we want to calculate the sum of two numbers. Before you write your sum method, make sure you can get the right input value from the keyboard. Did you cast the number from String to int or double? It's much faster and easier to solve small simple problems than large complex ones.

10. Write detailed comments

Writing comments can help other readers understand your ideas easily and quickly.